FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Abordagem de computação heterogénea para reamostragem e redimensionamento de vídeo de alto desempenho

José Pedro Soares João Pereira

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Jorge Manuel Gomes Barbosa

18 de Julho de 2018

Abordagem de computação heterogénea para reamostragem e redimensionamento de vídeo de alto desempenho

José Pedro Soares João Pereira

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Luís Paulo Reis
Arguente: Altino Sampaio
Orientador: Jorge Manuel Gomes Barbosa

18 de Julho de 2018

Resumo

A crescente popularidade dos canais de comunicação de televisão e plataformas de streaming motiva a publicação de novo conteúdo multimédia. De modo a alcançar e a agradar ao maior número de consumidores, este conteúdo deve ser criado o mais rapidamente possível e com a melhor qualidade disponível. Os conteúdos de multimédia de elevada qualidade são obtidos através da aplicação de várias operações de pós-produção. Visto que as operações de produção são aplicadas ao nível de cada pixel de uma imagem e imagens de alta qualidade são constituídas por uma quantidade elevada de pixeis, estas operações reclamam uma capacidade de recursos computacionais diretamente proporcionais ao tamanho das imagens a serem processadas.

Atualmente, devido ao progresso da tecnologia contida nas placas de processamento gráfico existe uma maior capacidade de processamento em relação à capacidade dos processadores, tipicamente, aliada a uma utilização mais eficiente de energia. A abordagem que combina as capacidades de processamento de um processador e de uma placa gráfica de uma máquina, a abordagem de computação heterogénea, é ideal para o desenvolvimento de software de alto desempenho.

O objetivo deste trabalho consiste em analisar e implementar uma abordagem de computação heterogénea utilizando as capacidades das placas gráficas para uma das fases de pós-produção, a fase de reamostragem e redimensionamento de vídeo sem compressão em tempo real. Especificamente, este projeto aborda os detalhes e peculiaridades da implementação de uma abordagem que alia as capacidades de processamento do CPU e GPU durante a fase de pós-produção referida de vídeo sem codificação, assegurando que o tempo de processamento de um vídeo é menor ou igual ao seu tempo de captura. A solução implementada utiliza a ferramenta OpenMP e a plataforma de desenvolvimento CUDA para paralelização e tirar partido das capacidades de processamento dos processadores e das placas gráficas.

Os resultados obtidos pela ferramenta desenvolvida mostram que a implementação da abordagem referida permitiu ganhos de desempenho entre 48% e 57% em relação à solução mais utilizada na área de multimédia para processamento e aplicação de operações de pós-produção de vídeo, a ferramenta FFmpeg.

A partir dos resultados obtidos por este trabalho é possível concluir-se que as placas gráficas são uma ferramenta com grandes capacidades computacionais que podem ser utilizadas para extrair paralelismo e potenciar o desempenho da aplicação de operações de pós-produção de vídeo profissional.

Abstract

The increasing popularity of communication channels such as television broadcasters and streaming platforms motivates the issue of new multimedia content. In order to reach and please the most customers, this content should be created as fast as possible with the best quality available. High quality multimedia content is obtained through the application of various post-production operations. Since post-production operations are applied at an image's pixels level and high quality images are constituted by an high quantity of pixels, these operations require computational resources directly proportional to their size.

Nowadays, due to the advance of the technology contained in the graphics processing units there is a greater processing capability in relation to the processing capability of processors, typically, conjugated with a better power usage efficiency. The approach that combines the processing capabilities of a processor and a graphics card of a machine, the heterogeneous computing approach, is ideal to the development of high performance software.

The goal of this work is to analyze and implement an heterogeneous computing approach using the capabilities of graphics cards during one of the post-production phases, the resample and resizing phase of uncompressed video in real time. Specifically, this project tackles the details and implementation peculiarities of an approach that allies the computing capabilities of CPU and GPU during the stated post-production phase of video without encoding, guaranteeing that the processing time of a video is less or equal than its capturing time. The implemented solution uses OpenMP and the development framework CUDA to parallelize and take advantage of the computing capabilities of processors and graphics cards.

The results of the developed tool show that the implementation of the stated approach allowed a performance gain between 48% and 57% in comparison to the most used solution in multimedia area to process and to apply video post-production operations, the FFmpeg tool.

From the obtained results it is possible to conclude that graphics cards are a tool with great computing capabilities that can be used to extract parallelism and potentiate the performance of the application of professional video post-production operations.

Agradecimentos

Quero agradecer ao meus pais por me ajudarem, por todo o apoio, esforço e conselhos que muito me ajudaram durante todo o meu percurso académico.

Aos meus amigos e colegas de trabalho, o meu obrigado por toda a camaradagem e momentos de boa disposição.

Agradeço, também, a todos os elementos e direção da Mog Technologies, em particular ao Alexandre Ulisses, Ivone Amorim e ao Pedro Santos o seu acompanhamento e conhecimento técnico que partilharam comigo durante esta fase académica.

O meu muito obrigado ao professor Jorge Barbosa por toda a paciência e tolerância que demonstrou na orientação desta dissertação.

Pedro Pereira

"The good thing about computers is that they do what you tell them to do.
The bad news is that they do what you tell them to do."

Ted Nelson

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

ALU	Arithmetic Logic Unit
API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
FHD	Full High Definition
FPS	Frames por Segundo
GPGPU	General Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
IIR	Infinite Impulse Response
MPI	Message Passing Interface
NAN	Not An Number
NN	Nearest Neighbor
UHD	Ultra High Definition
ULP	Unit of Least Precision
UML	Unified Modelling Language
RAM	Random Access Memory
SM	Streaming Multiprocessors
SP	Stream Processors

Capítulo 1

Introdução

A sociedade do século vinte e um tem sofrido uma drástica alteração cultural, económica e social devido à utilização das novas tecnologias dos dispositivos eletrónicos como computadores pessoais, smartphones, e tablets, que combinada à utilização da internet permite que qualquer indivíduo possa aceder, visualizar, modificar, armazenar e partilhar conteúdos de multimédia com um alcance virtualmente global. Por esta razão as indústrias de jornalismo, educação, entretenimento, a indústria editorial e de música têm sofrido significativas transformações para acompanhar a tendência, levando à utilização de meios de comunicação diferentes dos considerados convencionais antigamente de modo a atingir os seus objetivos, como a utilização de plataformas de streaming e de *video on demand*. Todo o mercado de criação de conteúdos multimédia, constituído maioritariamente pelas indústrias referidas, está direcionado para a publicação de novo conteúdo multimédia o mais rapidamente possível com a melhor qualidade disponível culminando numa vantagem sobre outros negócios do mercado.

Desde a invenção dos primeiros computadores, o poder de processamento e as capacidades de armazenamento de informação têm crescido exponencialmente. Com a evolução da tecnologia surgiram diferentes abordagens para realizar os processos de pós-produção de vídeo necessários para gerar conteúdos de multimédia de elevada qualidade. A computação paralela e computação heterogénea são duas abordagens que podem ser tidas em conta para a aplicação dos processos de pós-produção, pois estas abordagens permitem a execução concorrente dos mecanismos de um sistema. Utilizando as capacidades computacionais de vários tipos de unidades de processamento é permitido acelerar a aplicação das operações de pós-produção de vídeo que resulta em conteúdos multimédia com elevada qualidade processados num menor tempo.

Este trabalho foi desenvolvido no contexto da empresa MOG Technologies especializada no desenvolvimento de novas plataformas tecnológicas que fornecem produtos, sistemas e soluções inovadoras que automatizam os processos de trabalho dos operadores de conteúdos de multimédia profissional, nomeadamente, editoras e produtoras de vídeo. Um dos processos de trabalho automatizado é o processo de pós-produção de vídeo, automatizado através de uma solução de

ferramentas de *Ingest* que manipulam e transportam um vídeo desde um ponto de origem até a um ponto de destino da cadeia de produção.

De forma a obter um melhor desempenho das soluções desenvolvidas pela empresa e de modo a acompanhar as restrições cada vez mais rigorosas impostas pela indústria, os processos automatizados têm sido reestruturados de modo a utilizarem abordagens de computação paralela e computação heterogénea.

1.1 Motivação

O processo de reamostragem e redimensionamento de vídeo é um dos processos necessários a serem realizados durante a fase de pós-produção de conteúdos multimédia. A aplicação dos processos de pós-produção em vídeo sem compressão, isto é, em formato *raw*, é uma tarefa dispendiosa em termos de recursos computacionais.

As frames de um vídeo sem compressão são constituídas por um elevado número de pixeis. Uma imagem na resolução *Full High Definition*, ou *FHD*, com uma dimensão de 1920×1080 pixeis, é constituída, aproximadamente, por dois milhões de pixeis. Atualmente, são cada vez mais utilizadas resoluções superiores ao *FHD*, como é o caso do *Ultra High Definition*, ou *UHD*, designadamente as resoluções *4K* e *8K*, aproximadamente constituídas, respetivamente, por nove milhões e por trinta e cinco milhões de pixeis.

Atendendo que o processo de reamostragem e redimensionamento de vídeo sem compressão opera ao nível de cada pixel de uma frame, a capacidade de processamento necessária para efetuar operações sobre as frames é diretamente proporcional à sua resolução. De forma que os conteúdos de multimédia estejam disponíveis assim que possível, a aplicação dos processos de pós-produção deve ser realizada, idealmente, em tempo real. Isto é, o processamento das frames de um vídeo deve ser realizado num tempo menor ou igual ao tempo de captura do mesmo.

Por estes motivos, a aplicação dos processos de pós-produção, respeitando as restrições da indústria, torna-se cada vez mais difícil de ser efetuada para vídeos com resoluções mais elevadas sem que haja uma nova análise e uma nova implementação das soluções utilizadas até ao momento para o efeito.

1.2 Objetivos

Esta dissertação tem como objetivo a implementação de uma solução proprietária que realiza a operação de reamostragem e redimensionamento de vídeo com um desempenho equivalente ou melhor que a ferramenta utilizada atualmente nos produtos fornecidos pela MOG Technologies, a ferramenta FFmpeg.

A solução desenvolvida deve tirar o máximo proveito da heterogeneidade de processamento existente nos produtos da MOG Technologies, utilizando as capacidades computacionais do processador e da unidade de processamento gráfico das máquinas para proporcionar uma resposta escalável às crescentes resoluções de vídeo.

A solução proposta deve ser capaz de atingir o processamento em tempo real de vídeo em *stream* de elevadas resoluções e reduzir o tempo de execução de aplicação da operação de reamostragem e redimensionamento em vídeos previamente capturados.

Assim, as finalidades desta dissertação são a análise das técnicas utilizadas na aplicação da operação de reamostragem e redimensionamento da fase de pós-produção de conteúdos multimédia e a implementação de uma solução escalável para o problema, recorrendo a uma abordagem de computação heterogénea. Enumeram-se os seguintes objetivos deste trabalho:

- Analisar a arquitetura de unidades de processamento, as técnicas e as ferramentas utilizadas no desenvolvimento de programas paralelos;
- Revisão bibliográfica dos algoritmos de reamostragem e redimensionamento de vídeo, e dos modelos de cor e respetivos formatos de pixeis de representação de imagens;
- Implementar uma solução paralela eficiente para o processo de reamostragem e redimensionamento de vídeo.

1.3 Estrutura da Dissertação

Este documento é constituído por seis capítulos, incluindo este capítulo introdutório de contextualização do problema.

O capítulo 2 intitulado de "Programação Paralela" expõe os principais detalhes das arquiteturas das unidades de processamento utilizados em sistemas das áreas de computação paralela e computação heterogénea, descreve os principais conceitos da área de sistemas paralelos e analisa as ferramentas e plataformas de desenvolvimento utilizadas para implementar sistemas paralelos eficientes.

O capítulo 3 intitulado de "Processamento de Vídeo" apresenta as noções básicas de representação de imagens como o modelo de cor utilizado em vídeos digitais e os seus correspondentes tipos de formatos de pixeis, e descreve a operação de reamostragem e redimensionamento de vídeo considerando os diferentes algoritmos para o efeito. Neste capítulo, também, são abordadas as ferramentas de processamento de imagem e vídeo utilizadas para a aplicação do processo considerado.

O capítulo 4 intitulado de "Solução Proposta" descreve a solução desenvolvida do processo de reamostragem e redimensionamento de vídeo, e a lógica por de trás dos métodos implementados. Neste capítulo são referidas as técnicas utilizadas para a otimização do problema e possíveis limitações das mesmas.

O capítulo 5 intitulado de "Resultados" apresenta os resultados obtidos pela solução desenvolvida no trabalho desta dissertação para a operação de reamostragem e redimensionamento de vídeo em termos de tempo de execução e a justificação das peculiaridades encontradas em alguns casos de teste.

Por último, o capítulo 6 finaliza o trabalho desta dissertação referindo as conclusões assimiladas, os objetivos alcançados e o trabalho futuro associado à solução implementada nesta dissertação.

Capítulo 2

Computação Paralela

Neste capítulo é analisado o estado da arte referente à área de computação paralela e computação heterogénea, são analisadas as componente que constituem as unidades de processamento central e as unidades de processamento gráfico dos computadores atuais, estabelecendo uma comparação entre ambas. Como conclusão deste capítulo são apresentadas as diferentes ferramentas e plataformas de desenvolvimento utilizadas na atualidade para a implementação de sistemas de computação paralela e computação heterogénea.

2.1 Unidades de Processamento

Os computadores atuais seguem a arquitetura desenvolvida por John von Neuman [8], designada de máquina de von Neuman. Segundo esta arquitetura, uma máquina é dividida em três componentes principais como apresentado na figura 2.1, sendo estas: a unidade de processamento central, ou CPU, a memória de acesso aleatório, ou RAM, e a interface de entrada e saída.

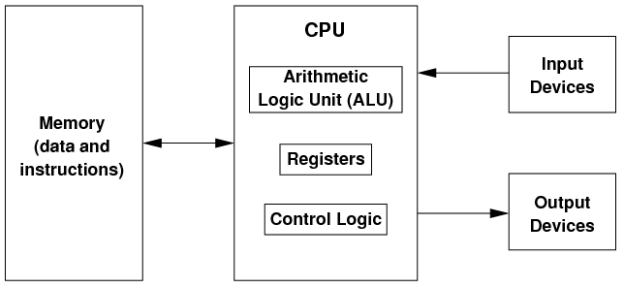

Figura 2.1: Diagrama de arquitetura de von Neuman [1].

A unidade de processamento central é a componente de um computador responsável pela execução de instruções de um determinado programa. O CPU é constituído pela unidade de lógica e aritmética, ou ALU, os registos do processador e uma unidade de controlo. A ALU é um circuito responsável pela execução de operações aritméticas e lógicas. Os registos do processador estão

encarregues de fornecer operandos à ALU e armazenar os respetivos resultados das operações. A unidade de controlo coordena a recuperação dos dados armazenados na memória e a execução das operações realizadas pela ALU, registos do processador e outras componentes. A RAM armazena as instruções a serem executadas e os dados necessários para as realizar. A interface de entrada e saída faz a ligação entre a RAM e o CPU, podendo também criar ligação com outro tipo de periféricos ou hardware. Os sistemas de computação paralela são constituídos por várias unidades de processamento que seguem a arquitetura de von Neuman o que permite executar em simultâneo múltiplas operações e conter diferentes dados armazenados [9].

2.1.1 Evolução dos CPUs

A lei de Moore, criada por Gordon Moore, co-fundador da Intel, é a observação que o número de transístores de um circuito integrado duplica a cada dois anos [10]. Os transístores permitem criar circuitos complexos. Quanto maior for o número de transístores presentes num circuito integrado, como por exemplo um processador, maior será o número dos circuitos complexos destinados a realizarem as operações aritméticas e lógicas, e por consequência maior será a capacidade de um processador de execução de instruções [2]. Por esta razão, a capacidade de processamento de um CPU está intrinsecamente relacionada com a lei de Moore.

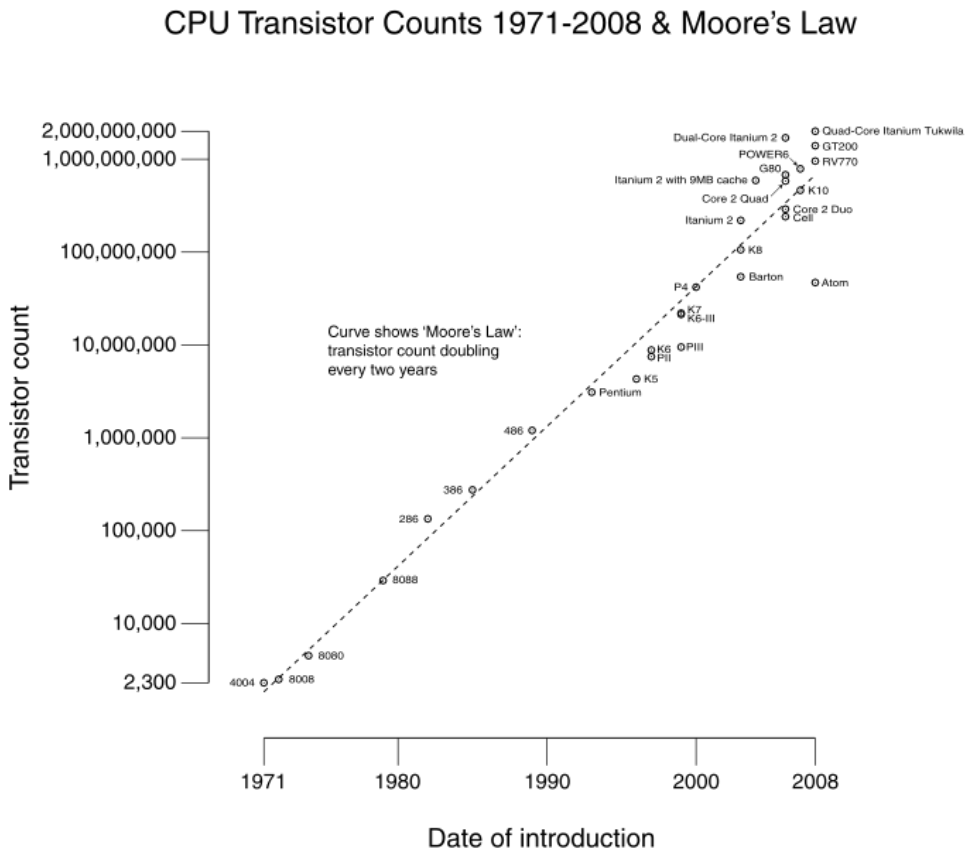

Figura 2.2: Relação entre o número de transístores dos processadores e a lei de Moore [2].

A utilização de um maior número de transístores implica um maior consumo de energia por parte de um processador. Até 2004, a abordagem utilizada para aumentar a capacidade de processamento de um CPU era a integração de um maior número de transístores no circuito, de modo a aumentar a frequência de relógio que dita a cadência de execução das instruções a serem realizadas. O aumento da frequência de um processador causa um maior consumo de energia devido à dissipação de calor. A cada aumento de 400 MHz na frequência de execução de instruções de um CPU, existe um incremento de 60% da energia consumida pelo processador [11]. Esta abordagem tornou-se inviável visto que o custo da energia consumida não compensa o ganho de desempenho do processador.

Por esta razão outro tipo de abordagem foi tida em conta, nomeadamente, a arquitetura *multi-core*. O aumento do número de núcleos de processamento de um processador permite ter uma frequência de relógio mais lenta e aumentar o número de instruções executadas [12]. Como exemplo considera-se uma certa frequência de relógio de um processador, um núcleo de processamento a 20% dessa frequência economiza 50% da energia necessária apenas sacrificando 13% da sua performance. Ao dividir o trabalho a ser realizado por dois núcleos de processamento a 80% de frequência de relógio, é possível atingir 43% de ganho de performance pelo mesmo custo energético [11].

2.1.2 Hierarquia de Memória - CPU

Na arquitetura de um computador, as componentes destinadas ao armazenamento de dados são ordenadas hierarquicamente com base no seu tempo de resposta a acessos [13]. A hierarquia de memória afeta o desempenho de um computador de tal modo que a implementação de um programa de alto desempenho deve considerar as capacidades e limitações de cada uma das componentes de memória utilizadas. As componentes de memória presentes num computador atual são apresentadas, por ordem crescente de tempo de resposta e capacidade de armazenamento:

- Registos do processador - Componente da arquitetura de um computador de acesso a memória mais rápido, cerca de um ciclo de relógio do CPU, mas de reduzida capacidade de armazenamento. Normalmente, esta componente de memória é utilizada para armazenar temporariamente os resultados de operações intermédias e valores de variáveis utilizadas frequentemente num processo;

- Memória cache - É a componente de memória mais próxima do processador que armazena fragmentos da memória principal. A maioria dos computadores têm diferentes níveis deste tipo de memória que diferem segundo a proximidade ao núcleo do processador, capacidade de armazenamento e tempo de resposta a acessos a memória. Este tipo de memória tem um sistema próprio de gestão que prioriza o armazenamento de dados utilizados frequentemente e liberta dados armazenados considerados obsoletos no contexto de um processo;

- Memória principal - Componente mais utilizada para conservação de dados devido à sua abundante capacidade de armazenamento. Contudo, a memória principal é a componente

de memória com maior latência de resposta a acessos a memória das componentes apresentadas;

- Memória Secundária - Este tipo de memória não está imediatamente disponível a ser utilizada pelo computador visto que necessita de interação humana para o efeito. O acesso a este tipo de memória não pode ser processado diretamente pelo CPU. Para aceder a este nível de memória é necessário copiar os dados armazenados na componente para memória principal. Discos compactos e memórias flash USB são dispositivos considerados memórias secundárias.

2.2 Taxonomia de Flynn

Atualmente, as arquiteturas dos computadores evoluíram para máquinas paralelas devido ao limite prático da frequência de relógio de um CPU em relação ao seu consumo de energia. A taxonomia de Flynn é uma classificação das arquiteturas dos computadores baseada no modo como as unidades de processamento funcionam, como a sua memória é organizada e como são realizadas as comunicações do processador [14]:

- SISD, ou *single instruction single data stream*, um único processador processa um único elemento dos dados por unidade de tempo. Este tipo de arquitetura encontra-se presente em micro-controladores e antigos computadores pessoais;

- SIMD, ou *single instruction multiple data streams*, uma mesma instrução é aplicada a múltiplos elementos de dados a cada unidade de tempo. Esta arquitetura é caraterística de processadores vetoriais e unidades de processamento gráfico;

- MISD, ou *multiple instructions single data stream*, um conjunto de unidades de processamento, conectadas sequencialmente, realizam diferentes operações sobre os mesmos dados. *Systolic arrays* são um exemplo de aplicação desta arquitetura;

- MIMD, ou *multiple instructions multiple data streams*, por unidade de tempo cada processador, contendo uma unidade de controlo independente, pode executar diferentes instruções de um programa. Esta arquitetura está presente na maioria das máquinas *multi-core* e multi-computadores.

2.2.1 Arquitetura *Multi-core*

Um processador *multi-core* segue a arquitetura MIMD, ou *multiple instructions multiple data stream*. Segundo esta arquitetura o CPU pode realizar múltiplas operações sobre diferentes dados e por isso é capaz de executar um programa em paralelo. Um processador multi-core é constituído por várias unidades de processamento contendo cada uma: unidades de lógica e aritmética, e unidades de controlo independentes.

2.2.2 Arquitetura *Many-core*

Um processador *many-core* segue a arquitetura SIMD, ou *single instruction multiple data streams*, é projetado para realizar processamento paralelo de menor consumo energético à custa de latência de comunicação entre os núcleos do processador, ou *core*, e de fraco desempenho singular dos processadores.

2.3 Unidades de Processamento Gráfico

Uma unidade de processamento gráfico, também designada de GPU, é um circuito especializado em alterar e manipular a memória de maneira a que o processamento de imagens a serem apresentadas num dispositivo de exibição seja acelerado. O GPU é extremamente independente em relação às restantes componentes de uma máquina por possuir a sua própria hierarquia de memória, mecanismos de transferência e de processamento de dados. As unidades de processamento gráfico são constituídas por múltiplos *streaming multiprocessors*, ou SM, que por sua vez são constituídos por múltiplos *stream processors*, ou SP.

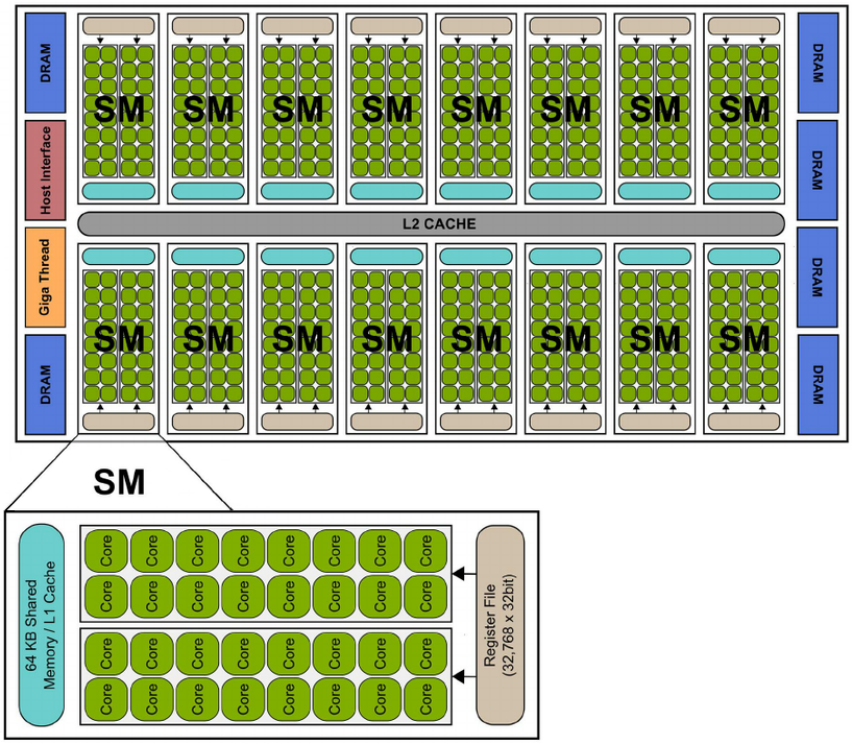

Figura 2.3: Composição e arquitetura interna de uma unidade de processamento gráfica [3].

Os *streaming multiprocessors* são unidades fundamentais de processamento dos GPUs. Os SMs, também designados de *blocks*, são constituídos por vários *stream processors* e uma unidade de controlo. A unidade de controlo é responsável por distribuir dados pelo conjunto de SPs. Esta unidade permite a execução síncrona da mesma instrução em todos os *stream processors*. Por esta razão, os *streaming multiprocessors* seguem a classificação SIMD de arquitetura.

Os *stream processors*, também designados de *cores* ou *threads*, são as unidades que constituem os SMs, utilizadas, efetivamente, no processamento das operações atribuídas ao GPU. Os SPs seguem a arquitetura SISD. Por conseguinte, cada *stream processor* executa uma instrução sobre um elemento em específico do conjunto de dados.

Devido à vasta quantidade de núcleos de processamento, à segregação dos mecanismos de acesso a memória e aos mecanismos de execução independente de instruções, os GPUs têm uma estrutura altamente paralela em comparação com os processadores, o que permite o processamento excecionalmente eficiente de blocos de informação em paralelo.

2.3.1 Hierarquia de Memória - GPU

A hierarquia de memória das unidades de processamento gráfico têm uma grande influência no desempenho das operações que esta executa. Uma estratégia de implementação que considere as diferentes caraterísticas dos múltiplos níveis de memória dos GPUs e dos seus mecanismos de gestão, permite tirar partido de um maior fluxo de dados e por consequência um melhor desempenho das operações realizadas nestas unidades.

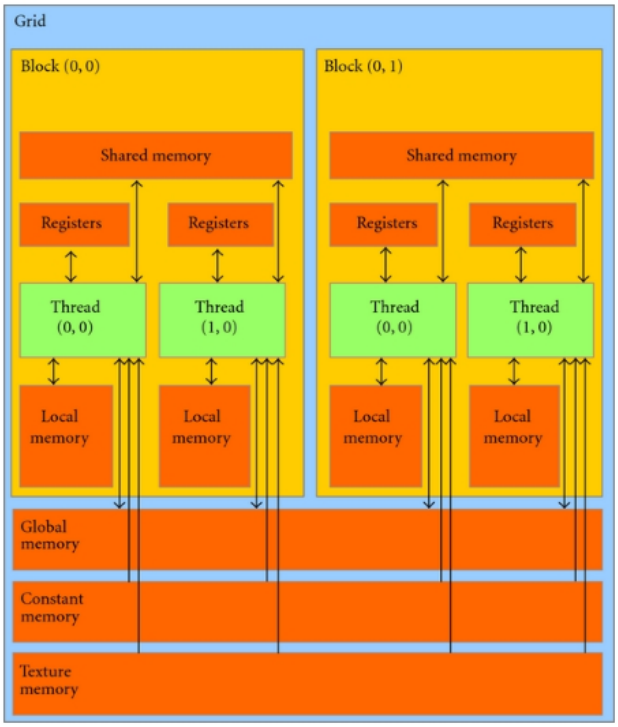

Figura 2.4: Disposição dos níveis memória em relação à arquitetura das unidades de processamento gráfico [4].

Na figura 2.4 é apresentado um diagrama simplificado da arquitetura de um GPU constituído apenas por dois SMs, por sua vez constituído por dois SPs, e as componentes de memória que constituem as unidades de processamento gráfico.

2.3.1.1 Memória Global

A memória global é a componente de armazenamento principal das unidades de processamento gráfico. Todo este nível de memória é endereçável e os dados neste armazenados são persistentes durante toda a atividade do GPU, sendo apenas libertados ou removidos por instruções explícitas ou pela desativação da placa gráfica.

Esta componente de memória apresenta a maior capacidade de armazenamento do GPU em relação aos outros níveis de memória. Qualquer *streaming multiprocessor* do GPU pode aceder a este nível de memória e, por consequência, todos os SPs. Os acessos a este tipo de memória são geridos através de um sistema de cache de dois níveis, designados de cache L1 e L2. Existe uma instância de cache L1 presente em cada SM e apenas uma instância de cache L2 partilhada por todos os SMs.

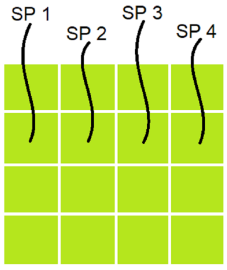

Figura 2.5: Acesso coalescido a memória pelos *stream processors*.

As respostas de acessos a memória global são serializadas pelo GPU para diminuir o tempo de transferência de dados. Quando um *stream processor* realiza um acesso a memória global toda a região circundante à posição desse acesso é serializada e transferida para o sistema de cache [15]. Para tirar o máximo proveito desta funcionalidade, o acesso a memória global de um GPU é mais eficiente, e com menor latência de resposta de acessos a memória, quando cada SP acede a posições de memória consecutivas à posição de memória acedida pelos SPs vizinhos, como pode ser esquematizado pela figura 2.5. Este modelo de acesso a memória é denominado de acesso a memória coalescida, ou *coalesced memory access*.

2.3.1.2 Memória Local

A memória local encontra-se ao mesmo nível da memória global por apresentar valores de latência de resposta de acessos a memória similares. Cada componente de memória local está associada a apenas um SP e, por essa razão, é considerada independente das restantes. Este nível de memória é utilizado pelo GPU quando um *stream processor* não tem capacidade para armazenar em registos os valores das variáveis utilizadas nas suas operações. Os dados armazenados na memória local são conservados durante o tempo de execução de uma operação por parte do SP correspondente.

2.3.1.3 Memória Constante

A componente de memória constante encontra-se ao mesmo nível da memória global e, à semelhança da última, todo este nível de memória é endereçável pelos SMs e persistente durante toda a atividade do GPU. A memória constante tem o seu próprio sistema de cache que difere do sistema de cache da memória global. Todos os *stream processors* de um SM têm acesso à componente de memória constante, embora apenas com permissões de leitura e não de escrita. Os dados armazenados na memória constante são definidos pelo CPU antes das operações a serem executadas pelo GPU.

O acesso a memória constante é extremamente rápido em comparação com os acessos a memória global, isto se todos os SPs de um SM acederem ao mesmo índice de memória. A latência das respostas de acessos a este nível de memória é reduzido, tendo um desempenho semelhante ao apresentado por níveis de memória mais próximos dos *cores* [16]. Esta peculiaridade deve-se a simplicidade do sistema de gestão de memória desta componente por não necessitar da implementação de mecanismos de escrita da memória.

2.3.1.4 Memória de Textura

À semelhança da memória global e constante, a memória de textura pode ser acedida por todos os SMs de um GPU e encontra-se ao mesmo nível que as primeiras. A componente de memória de textura tem o seu próprio sistema de cache, embora apenas disponibilize permissões de leitura ao GPU enquanto a escrita dos dados nela armazenados é da responsabilidade do CPU.

O sistema de cache desta memória é ideal para operações que realizam acessos a memória não coalescidos, o que torna este nível de memória uma opção eficaz para a impossibilidade da utilização eficiente da memória global [17].

O sistema de cache da memória de textura prioriza a localidade espacial dos acessos a memória realizado pelos *stream processors*. Isto é, o modelo de acesso coalescido compele o acesso a posições de memória consecutivas por parte de SPs consecutivos, enquanto, este sistema de cache permite o acesso eficiente de um *stream processor* a qualquer posição de memória desde que os dados acedidos pelo último se encontrem próximos de dados previamente acedidos por outros SPs.

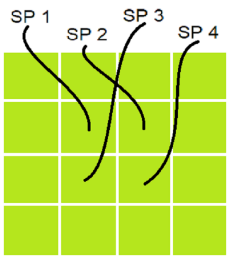

Figura 2.6: Acesso eficiente à memória de textura pelos *stream processors*.

O padrão de acesso descrito é caraterizado pela prioridade da localidade espacial dos dados, ou *spatial locality*, e pode ser representado pelo diagrama da figura 2.6.

2.3.1.5 Memória Partilhada

A memória partilhada é uma componente de rápida resposta a acessos de memória por se encontrar integrada numa região próxima dos *stream processors* do GPU. Este nível de memória é integrada em cada *streaming multiprocessor* de modo a possibilitar o acesso e transferências de dados entre os diferentes SPs que o constituem.

Para armazenar dados da memória global nesta componente e tirar partido da reduzida latência de resposta de acessos a memória partilhada, é necessário que a operação realizada em cada SP tenha explicitamente uma operação de leitura ou escrita de dados, seguindo um acesso coalescido, da memória global para este nível de memória [16].

A latência de resposta de acessos a memória partilhada é menor quando todos os *stream processors* acedem a diferentes bancos de memória (uma posição de quatro bytes de memória) ou ao mesmo endereço de memória.

2.3.1.6 Registos

Os registos são a componente de memória mais próxima dos SPs. Cada SP tem uma memória de registos própria que é independente dos restantes *stream processors*. Por estas razões, os registos são a componente de memória mais rápida das unidades de processamento gráfico [16], isto é, com menor latência de resposta de acessos de memória.

Os valores das variáveis utilizadas durante a execução de uma operação, por parte de um SP, são armazenados nos registos correspondentes ao *stream processor*. Devido à reduzida capacidade de memória dos registos, caso não seja mais possível armazenar dados nesta componente, a memória local passará a ser utilizada. Os dados armazenados nos registos apenas são mantidos durante a atividade do SP correspondente.

Tabela 2.1: Visão geral dos níveis de memória do GPU ordenada de forma decrescente em relação à latência de respostas de acessos a memória.

Memória	Permissões[1]	Acessível Por	Cache	Observações
Global	L / E	SP, SM e CPU	Sim	Ampla capacidade de armazenamento
Local	L / E	SP	Sim	Reserva de armazenamento dos registos
Constante	L	SP, SM e CPU	Sim	Rápido acesso ao mesmo endereço
Textura	L	SP, SM e CPU	Sim	Localidade espacial de acesso
Partilhada	L / E	SP, SM	Não	Latência de transferências reduzida
Registos	L / E	SP	Não	Acesso virtualmente instantâneo

[1] L/E - Leitura e Escrita; L - Leitura; E - Escrita;

2.4 CPU vs GPU

Em comparação, os processadores modernos são capazes de realizar processamento em paralelo de várias operações sobre múltiplos dados, de acordo com a arquitetura MIMD que seguem. Enquanto as unidades de processamento gráfico seguem a arquitetura SIMD, implicando que estes dispositivos de processamento realizem apenas uma operação sobre múltiplos dados em paralelo.

A área de superfície dos circuitos dos CPUs é ocupada, maioritariamente, pelas unidades de controlo e o sistema de cache, restando apenas uma pequena área reservada a unidades de lógica e aritmética. Por outro lado, a área de superfície dos circuitos dos GPUs é constituída, essencialmente, por unidades de lógica e aritmética e apenas uma pequena região é dedicada às unidades de controlo e do sistema de cache.

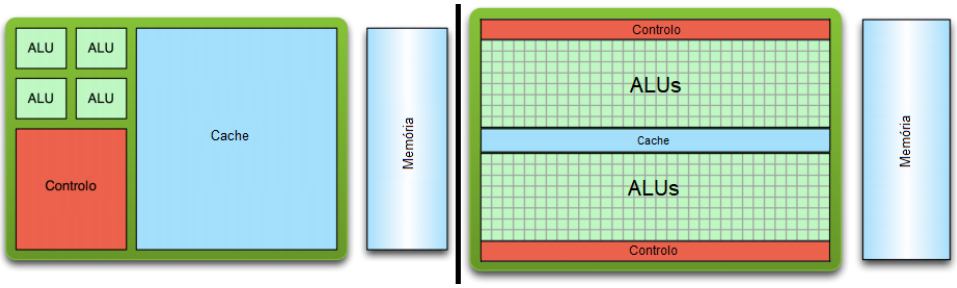

Figura 2.7: Diferenças da arquitetura interna entre um CPU (esquerda) e um GPUs (direita).

As diferenças arquiteturais destas unidades de processamento têm um impacto direto na forma como executam as suas operações. O GPU é mais restrito em termos de instruções executadas do que o CPU, mas tem um maior potencial para realizar operações numéricas e executar problemas de processamento de dados. Outra diferença técnica entre estes dois tipos de processadores é a ampla largura de banda das unidades de processamento gráfico em relação ao CPU. Esta diferença é justificada pela presença dos sistemas de cache especializados das componentes de memória do GPU que permite a utilização eficiente dos seus vários níveis de memória. Os sistemas de cache do GPU levam a uma menor latência de transferência de dados entre os seus núcleos de processamento e, por consequência, a um melhor desempenho de execução.

Os processadores mantêm o balanço entre o seu poder computacional e o propósito geral de execução das diferentes funções que realizam em paralelo, enquanto as unidades de processamento gráfico proporcionam o máximo de poder computacional sofrendo das restrições que esse poder implica. A relação entre o CPU e o GPU é um balanço entre a flexibilidade de funcionamento e a capacidade computacional. Cada uma destas diferentes unidades de processamento apresenta uma melhor eficácia na resolução de problemas em relação à outra, dependendo do problema em questão. Devido às diferenças de arquitetura entre os CPUs e os GPUs é possível concluir que o CPU é uma unidade de processamento mais eficaz para processamento de múltiplas operações diferentes em paralelo relativamente ao GPU. Por outro lado, o GPU apresenta um melhor desempenho que o CPU na execução de problemas de processamento de múltiplos dados em paralelo por uma operação.

2.4.1 Computação Heterogénea

Computação heterogénea refere-se a sistemas paralelos que utilizam mais que um tipo de unidades de processamento. Este tipo de sistemas têm alto nível de desempenho e eficiência de consumo de energia adicionando processadores especializados para diferentes tipos de objetivos. A arquitetura de sistemas que implementam uma abordagem de computação heterogénea é, habitualmente, constituída por diferentes modelos de processadores, ou por CPUs e unidades de processamento gráfico. Este tipo de sistemas encarrega a execução de tarefas de computação paralela intensiva ao GPU enquanto o resto do programa é atendido e executado pelo CPU.

2.5 Programação Paralela

Computação paralela é a designação atribuída ao mecanismo segundo o qual vários processadores executam ou processam uma determinada operação em simultâneo. A utilização de computação paralela permite realizar computações complexas dividindo a carga de trabalho por mais de uma unidade de processamento, todas as unidades de processamento realizam as operações ao mesmo tempo [18].

De modo a ser possível dividir um problema em diferentes partes, é essencial identificar o tipo de problema a ser executado antes da formulação da solução paralela. Se P_D é um problema de domínio D e P_D é paralelizável, então D pode ser decomposto em k sub-problemas:

$$D = d_1 + d_2 + ... + d_k = \sum_{i=1}^{k} d_i \tag{2.1}$$

P_D é um problema de paralelização de dados se D é composto por elementos de dados e o problema é solucionado através da aplicação de uma determinada função $f()$ a todo o domínio:

$$f(D) = f(d_1) + f(d_2) + ... + f(d_k) = \sum_{i=1}^{k} f(d_i) \tag{2.2}$$

P_D é um problema de paralelização funcional se D é composto por diferentes operações e o problema é solucionado através da aplicação de cada operação a um mesmo conjunto de dados S:

$$D(S) = d_1(S) + d_2(S) + ... + d_k(S) = \sum_{i=1}^{k} d_i(S) \tag{2.3}$$

Problemas de paralelização de dados são ideais para serem solucionados recorrendo às capacidades computacionais do GPU. A arquitetura SIMD do GPU é ideal para a resolução de problemas cuja solução é obtida através da aplicação da mesma instrução em múltiplos fragmentos de dados. Por outro lado, problemas de paralelização funcional são ideias para serem solucionados recorrendo às potencialidades do CPU, pois a arquitetura MIMD do CPU permite a execução eficiente de múltiplas tarefas diferentes em cada núcleo de processamento, ou *core*.

2.5.1 Modelos de Programação paralela

De modo a tirar partido das capacidades computacionais das unidades de processamento atuais e a implementar uma solução paralela eficaz de um problema, é necessário haver uma abstração da arquitetura de memória da plataforma de destino e dos detalhes técnicos das unidades de processamento utilizadas.

Os modelos de programação paralela, o modelo de memória partilhada e o modelo de memória distribuída, fornecem paradigmas de implementação de soluções paralelas em que é possível a abstração das condições referidas [19]. Os modelos de programação paralela descrevem a interação entre a memória de uma máquina e as suas unidades de processamento.

2.5.1.1 Memória Partilhada

Segundo o modelo de memória partilhada, os processos que estão a ser executados nos núcleos de processamento, ou *threads*, interagem por variáveis declaradas no espaço de memória que partilham entre si. Com este modelo, os *threads* podem executar assincronamente sobre o mesmo conjunto de dados não havendo a necessidade de sincronização entre os mesmos.

Contudo poderá ser necessária a implementação de uma secção crítica com mecanismos de controlo de leitura e escrita caso algum dos *threads* necessite de acesso exclusivo à memória. No modelo de memória partilhada cada *thread* é constituído pelo seu próprio estado interno e as variáveis globais partilhadas, definidas pelo *thread* principal.

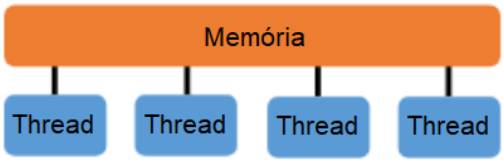

Figura 2.8: Modelo de memória partilhada.

Quando um dos *threads* acede um espaço da memória, provoca que a região de memória circundante a esse local, designado de linha de cache, seja copiado para o sistema de cache do CPU. Referências e acessos subsequentes ao mesmo espaço de memória ou a dados da mesma linha de cache podem ser realizados sem efetuar um acesso à memória principal do sistema.

A linha de cache é preservada no sistema de memória cache até que o sistema determine que é necessário restituir a coerência entre as duas memórias. Alterações de elementos de uma linha de cache, realizadas por diferentes *threads*, invalida toda essa linha. A cada alteração a linha é marcada como inválida e todos os outros processos que a contêm recebem o mesmo estatuto. O sistema obriga que estes processos recuperem a versão mais recente da linha invalidada a partir da memória principal de modo a manter a coerência dos dados entre os *threads*.

Esta situação é denominada de *false sharing* e ocorre frequentemente em soluções baseadas no modelo de memória partilhada. O *false sharing* provoca um aumento do tráfico de comunicações dos *threads* e um maior tempo de transferência de dados, o que diminui consideravelmente o desempenho de um sistema [20].

2.5.1.2 Memória Distribuída

No modelo de memória distribuída cada *thread* tem uma componente de memória independente dos restantes. Cada memória de um *thread* contém uma cópia dos dados da tarefa a ser executada. Os *threads* interagem entre si através de um sistema de mensagens utilizando a rede interna de comunicações do processador e, por essa razão, surge a necessidade de realizar operações de sincronização da execução de cada *thread* sobre os dados utilizados.

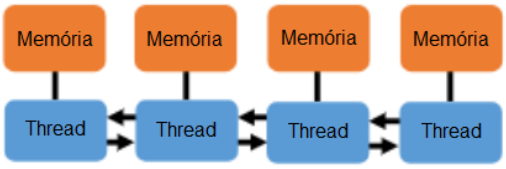

Figura 2.9: Modelo de memória distribuída.

2.5.2 Extração de Paralelismo

Ian Foster sugeriu uma metodologia utilizada para extrair paralelismo de um determinado problema e planear uma solução para o mesmo, maximizando o número de opções viáveis para o fazer e diminuindo o custo de retroceder na fase de desenvolvimento para resolver possíveis problemas de arquitetura da solução que possam surgir.

Esta metodologia implica um maior esforço na fase inicial do processo de modelação de uma solução e criar uma abstração a problemas como a compatibilidade de tecnologias, concorrência de execução da implementação e caraterísticas das componentes da máquina onde será integrada a solução [5]. A metodologia divide o processo de planeamento da arquitetura da solução em quatro fases distintas: a fase de divisão em partes ou *partitioning*, comunicação, aglomeração e mapeamento.

2.5.2.1 Divisão em Partes ou *Partitioning*

A fase de divisão em partes refere-se à decomposição em tarefas de dimensão inferior das atividades computacionais e dos dados do problema em questão. A decomposição das atividades computacionais a serem realizadas em tarefas disjuntas e dos dados é designada de decomposição funcional e decomposição do domínio/dados, respetivamente.

Figura 2.10: Fase de divisão do problema em partes [5].

2.5.2.2 Comunicação

A fase de comunicação concentra-se na criação do fluxo de informação entre as unidades de processamento e na coordenação da execução das tarefas criadas durante a fase de divisão do problema em partes, através da instauração de canais de comunicação entre as diferentes divisões. A especificidade do problema e o método de decomposição realizado determina o padrão de comunicação entre as tarefas e do programa paralelo.

Figura 2.11: Fase de criação dos canais de comunicação [5].

Este aspeto é usualmente qualificado de granularidade do problema. Um problema de granularidade fina pode ser dividido num elevado número de pequenas tarefas, enquanto um problema de granularidade grossa é dividido num número reduzido de tarefas de grande dimensão. Problemas de granularidade fina têm um alto potencial de serem paralelizáveis e apresentam uma maior latência de comunicações. Quanto aos problemas de granularidade grossa, os últimos têm uma solução fracamente paralelizável mas apresentam uma menor latência de comunicações.

2.5.2.3 Aglomeração

A fase de aglomeração refere-se à fase de planeamento do sistema paralelo em que as tarefas idealizadas durante a fase de *partitioning* são agrupadas de modo a que a carga de trabalho necessária para as executar compensa a ocupação de uma unidade de processamento.

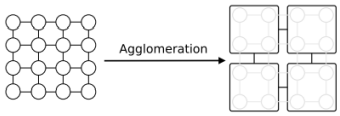

Figura 2.12: Fase de aglomeração de tarefas por unidades de processamento [5].

2.5.2.4 Mapeamento

A fase de mapeamento descreve o mecanismo de distribuição dos grupos de tarefas resultantes da fase de aglomeração pelas unidades de processamento disponíveis. O mapeamento é a última fase da metodologia de Foster e pode ser conseguida através de várias estratégias. O objetivo desta fase é encontrar a melhor distribuição da carga de trabalho pelas unidades de processamento minimizando a latência de comunicações entre as últimas.

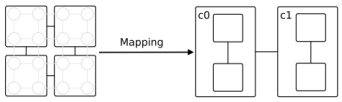

Figura 2.13: Fase de mapeamento de grupos de tarefas às unidades de processamento [5].

2.6 Ferramentas e Plataformas de Desenvolvimento

Na seguinte secção são apresentadas, descritas e comparadas as ferramentas e plataformas de desenvolvimento mais frequentemente utilizadas na atualidade para a implementação de sistemas que implementam mecanismos de computação paralela e computação heterogénea.

2.6.1 OpenMP

A ferramenta OpenMP é uma API, ou *application programming interface*, multi-plataforma para as linguagens de programação C, C++ e Fortran. A utilização desta ferramenta permite paralelizar, através de um mecanismo de *multi-threads*, uma determinada solução.

Multi-thread é o mecanismo que permite a utilização das capacidades de processamento de todos os cores de um CPU. Com *multi-threads* é possível realizar diversas tarefas concorrentemente com total suporte do sistema operativo [6].

O OpenMP é uma ferramenta destinada ao modelo de programação paralela de memória partilhada implementando o paralelismo do sistema através de diretivas de compilador *pragma* recorrendo ao mecanismo de execução paralela designado de *fork-join*.

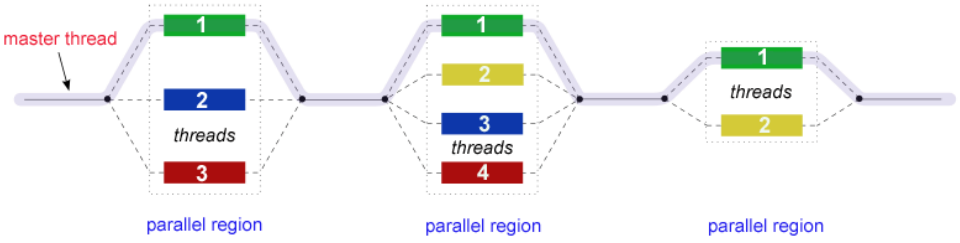

Figura 2.14: Mecanismo de paralelização *fork-join* utilizado pela ferramenta OpenMP [6].

De acordo com o mecanismo *fork-join*, as diretivas de compilador presentes nas instruções do sistema indicam regiões paralelas que serão executadas por diversos *threads*. No começo de um programa paralelo, implementado com OpenMP, apenas um *thread* está ativo, também designado de *master thread*. O *master thread* ativa regiões paralelas de instruções a serem executadas por várias unidades de processamento, sempre que as diretivas de compilador assim o indiquem.

2.6.2 *Message Passing Interface*

O *Message Passing Interface*, ou MPI é um standard que especifica os mecanismos de comunicação de sistemas paralelos que implementem o modelo paralelo de memória distribuída. O MPI detalha os métodos necessários para efetuar operações de comunicação e de controlo entre os vários processos.

Seguindo este standard é possível criar uma rede de unidades de processamento independentes com componentes de memória individuais capazes de executarem assincronamente diferentes operações.

2.6.3 OpenCL

O OpenCL é uma plataforma de desenvolvimento de software para a implementação de sistemas de computação heterogénea que aliam as capacidades computacionais de diferentes tipos de unidades de processamento.

A plataforma OpenCL fornece uma interface de métodos necessários para a implementação de sistemas paralelos cujo problema é a paralelização de dados da solução. O OpenCL pode ser executado em sistemas operativos baseados em Unix e Windows, e integrado com as linguagens de programação C e C++.

Esta plataforma de desenvolvimento permite a criação de métodos que se baseiam em tarefas, designados de *kernels*, e a execução dos mesmos sobre diferentes dados por unidade de processamento. Cada *kernel* é colocado numa fila de espera de operações a serem realizadas e são executados logo que uma das unidades de processamento tenha disponibilidade para o fazer.

2.6.4 *Compute Unified Device Architecture*

A *Compute Unified Device Architecture*, ou CUDA, é uma plataforma de computação paralela criada pela NVidia para o desenvolvimento de sistemas paralelos recorrendo a aceleração gráfica.

A plataforma CUDA permite tirar partido das capacidades de processamento das placas gráficas para a execução de programas de propósito geral, do tipo *General Purpose computing on Graphics Processing Units*, ou GPGPU. Este tipo de designação carateriza sistemas paralelos que utilizam as unidades de processamento gráfico para operações de processamento que usualmente são realizadas pelo CPU. Estes sistemas analisam e processam dados como se de uma imagem ou outro tipo de componentes gráficos se tratasse. Este paradigma atribui tarefas de intensa computação paralela ao GPU enquanto o restante programa é executado pelo CPU.

A plataforma CUDA pode ser executada em sistemas operativos baseados em Unix e Windows através das linguagens de programação C, C++ e Fortran, logo que estes sistemas contenham uma unidade de processamento gráfico da NVidia.

Em relação ao modo de funcionamento desta plataforma, a atividade da unidade de processamento gráfico é inicializada pelo CPU. O GPU opera sobre dados previamente transferidos a partir da memória da máquina para uma das componentes de memória do GPU. Após o processamento paralelo dos dados, os resultados são transferidos da memória do GPU para a memória da máquina. Este processo não bloqueia as capacidades de processamento do CPU, permitindo o último realizar outro tipo de operações enquanto o GPU se encontra em atividade. Por essa razão, esta plataforma permite tirar o máximo partido das capacidades computacionais de ambas componentes de processamento caso a solução seja corretamente implementada [21].

2.6.5 Comparação Entre Ferramentas e Plataformas de Desenvolvimento

Considerando a abordagem de computação heterogénea desta dissertação é necessário aferir quais das ferramentas e plataformas de desenvolvimento trazem mais vantagens para as unidades de processamento utilizadas na solução do problema.

A paralelização do CPU pode ser efetuada através da implementação de *threads* nativos. Contudo, esta abordagem é mais trabalhosa pois todas as funcionalidades têm de ser desenvolvidas considerando as particularidades dos vários sistemas operativos onde a solução pode ser executada. Tendo em conta a redução da complexidade do código da solução foi escolhida a ferramenta OpenMP pois a sua utilização requer apenas a indicação de diretivas de compilador *pragma* e é suportada pela maioria dos processadores utilizados na atualidade [6].

É possível verificar no seguinte excerto de código a fácil implementação de paralelização do CPU sem alterar drasticamente o código original, através da utilização do OpenMP. A inserção da diretiva de compilador permite que o bloco de instruções incluído no ciclo *for* seja dividido e executado pelos vários *threads* da máquina, introduzindo paralelismo de dados:

```
1 #pragma omp parallel for
2 for(int index = 0; index < N_ITERATIONS; index++){
3     foo();
4 }
```

O standard MPI é uma ferramenta versátil para o desenvolvimento de soluções paralelas de memória distribuída, especialmente quando esta solução recorre às capacidades de processamento de um grupo de diferentes máquinas, pois permite um mecanismo simples de transmissão de dados por sistemas com caraterísticas diferentes. Ainda assim, esta ferramenta não se enquadra com os objetivos desta dissertação, pois pretende-se implementar uma solução possível de ser executada numa só máquina.

O OpenCL e o CUDA são plataformas bastante semelhantes em termos de implementação de uma solução paralela, isto porque ambas necessitam da implementação de um *kernel*, utilizado para o processamento de um conjunto de dados. Ambas plataformas apresentam funcionalidades idênticas e, por essa razão, tornam o processo de portabilidade de soluções entre estas plataformas simples de ser realizado.

A plataforma CUDA é destinada à implementação de soluções que utilizem as unidades de processamento gráfico da marca NVidia. Por essa razão, o CUDA apresenta um melhor desempenho de execução em todos os aspetos em comparação com a plataforma OpenCL, no caso de utilização de um dos GPUs da marca [22]. Os aspetos referidos referem-se ao tempo de transferência de dados entre a memória da máquina e a componente de memória da unidade de processamento gráfico, e o tempo de execução de operações de processamento semelhantes.

2.7 Conclusão

A partir do conteúdo apresentado neste capítulo podemos aferir que as diferenças das arquiteturas entre um CPU e um GPU têm implicações na sua performance. A arquitetura de um processador, devido aos níveis de memória que contém, apresenta um valor baixo de latência de acessos a memória principal por possuir um sistema de memória cache que armazenam espaços de memória frequentemente acedidos. Os valores de tempo de acesso a memória só é afetado caso existam ocorrências de *false sharing* [20].

O GPU, por conter um número reduzido de unidades de controlo, permite a incorporação de um maior número de unidades de lógica e aritmética, o que possibilita trocar toda a flexibilidade funcional de processamento de um CPU por uma maior capacidade computacional.

A utilização de uma abordagem de computação heterogénea, que alia as capacidades de processamento do CPU e do GPU para a implementação deste trabalho, deve-se à fácil instalação da solução desenvolvida na maioria das máquinas utilizadas atualmente na aplicação dos processos de pós-produção. Tendo em conta esta abordagem, as ferramentas que apresentam mais vantagens de utilização são a ferramenta OpenMP e a plataforma de desenvolvimento CUDA.

A ferramenta OpenMP apresenta uma boa performance de execução para sistemas paralelos e é ideal para a paralelização funcional das instruções executadas no CPU sem introduzir um nível de complexidade adicional ao código do programa. A plataforma de desenvolvimento CUDA destaca-se por apresentar um melhor desempenho em termos gerais em relação à plataforma OpenCL [22]. Também, a solução resultante deste trabalho será destinada a ser executada em máquinas com unidades de processamento gráfico da NVidia, o que torna a utilização do CUDA ideal nestas circunstâncias.

Capítulo 3

Processamento de Vídeo

Este capítulo aborda o mecanismo de representação de vídeos e os seus respetivos detalhes. O processo de reamostragem e redimensionamento de vídeo é analisado tendo em conta os diferentes algoritmos para o efeito e os seus resultados. Como conclusão deste capítulo são enunciadas ferramentas de processamento de imagem que são utilizadas atualmente para realizar este tipo de operações em meios de pós-produção de vídeo profissional.

3.1 Estrutura de um Vídeo

Um vídeo é uma sequência de imagens, usualmente designadas de frames. Cada frame é uma imagem estática que visualizada em sequência, em conjunto com outras frames, recriam uma imagem animada do vídeo. O número de frames que constituem um vídeo depende do número de imagens capturadas por segundo na sua gravação. Este valor refere-se à métrica frame *rate* e representa o número de frames apresentadas por segundo, ou FPS, na reprodução de um vídeo. A frame *rate* também pode ser designada de frame *frequency* e ser expressa em hertz ao invés de frames por segundo.

Na indústria de multimédia é considerado um valor de frame *rate* normalizado entre vinte e quatro, e trinta e um frames por segundo para captura digital de vídeos [23]. À semelhança das frames de um vídeo, a visão humana captura imagens de cenários reais. A perceção de movimento da visão humana é criada por um processo de desfocagem das caraterísticas entre imagens consecutivas. Este processo cria a sensação de continuidade de movimento entre duas imagens.

O valor mínimo de frames por segundo em que o processo de desfocagem acontece entre duas imagens é de vinte e quatro frames por segundo [24]. Um valor menor que vinte e quatro frames por segundo retira a perceção visual humana de continuidade entre imagens, enquanto um maior valor implica maiores custos de pós-produção do vídeo devido a este ser constituído por um maior número de imagens.

A aplicação de operações de pós-produção de um vídeo, como o processo de reamostragem e redimensionamento pode ser tomada como a aplicação do mesmo processo à sequência de imagens que constituem o vídeo. Por esta razão, a resolução do problema de processamento de um vídeo por uma operação é considerado como a resolução do mesmo problema numa só imagem.

3.1.1 Representação de uma Imagem

As imagens podem ser distinguidas segundo o mecanismo de armazenamento gráfico utilizado na sua representação digital por imagens vetorizadas e imagens rasterizadas.

As imagens vetorizadas são geradas a partir de formulações matemáticas que definem primitivas geométricas, as suas posições relativas em relação a outras primitivas e as suas respetivas cores. As primitivas geométricas que constituem as imagens vetorizadas são representadas por pontos, linhas, curvas e formas geométricas básicas. A disposição e a escala destas primitivas são calculadas em função da resolução em que a imagem será apresentada. Por essa razão, é possível apresentar este tipo de imagens com diferentes dimensões sem que haja alteração da qualidade das suas caraterísticas.

Enquanto, as imagens rasterizadas são representadas digitalmente a partir de uma matriz de duas dimensões que armazena os valores de intensidade de cor para cada um dos pontos da imagem.

Como as imagens vetorizadas são constituídas a partir de primitivas geométricas, este tipo de imagens são inviáveis para a representação de imagens capturadas a partir de um cenário real por não existir um processo fiável de representação de imagens reais utilizando imagens vetoriais. Tendo em conta o contexto do problema desta dissertação serão apenas tidas em conta imagens rasterizadas.

3.2 Modelos de cor

Os pixeis de uma imagem representam o valor de intensidade de cor de um ponto de uma imagem segundo um modelo de cor específico. Um modelo de cor descreve de forma normalizada como o valor de intensidade de cor é representado utilizando um sistema de coordenadas de três dimensões segundo o qual cada valor de intensidade de cor é representado por um único ponto desse sistema. Os modelos de cor estão divididos em duas diferentes categorias: os modelos de cor específicos para representação de imagens e da qualidade das suas caraterísticas, como os modelos RGB, CMY, HSI; e os modelos de cor específicos para a representação de imagens de um vídeo focando-se na redução do tamanho necessário para a sua representação em detrimento das qualidade das caraterísticas das suas imagens, como os modelos YIQ, YUV, Y'CbCr.

No contexto do trabalho desta dissertação são apenas considerados os modelos YUV e Y'CbCr, pois os últimos são considerados standards quanto aos modelos de cor utilizados nos dispositivos de reprodução de vídeo. As designações dos modelos de cor YUV e Y'CbCr são consideradas permutáveis neste trabalho, visto que ambos modelos são caraterizados pelas mesmas componentes de cor de uma imagem e diferem apenas para que tipo de dispositivo de reprodução de vídeos

são direcionados. O modelo YUV é destinado para dispositivos de reprodução de vídeo analógico, enquanto o modelo Y'CbCr é direcionado para dispositivos de reprodução de vídeo digital [25].

As diferenças entre os dispositivos de reprodução de vídeo analógico e digital são mais relevantes em relação à representação de cores de uma imagem. Tendo em conta os diferentes dispositivos alvo dos modelos de cor YUV e Y'CbCr, ambos têm diferentes valores de coeficientes no cálculo do valor das suas amostras de crominância. Nas equações seguintes é possível observar as diferenças dos valores das componentes dos modelos de cor YUV e Y'CbCr a partir do modelo de cor RGB [26], respetivamente:

$$Y = 0.299 \times R + 0.587 \times G + 0.114 \times B \tag{3.1}$$
$$U = -0.147 \times R - 0.289 \times G + 0.436 \times B \tag{3.2}$$
$$V = 0.615 \times R - 0.515 \times G - 0.1 \times B \tag{3.3}$$

$$Y' = 0.299 \times R + 0.587 \times G + 0.114 \times B \tag{3.4}$$
$$Cb = -0.169 \times R - 0.331 \times G + 0.499 \times B + 128 \tag{3.5}$$
$$Cr = 0.499 \times R - 0.418 \times G - 0.0813 \times B + 128 \tag{3.6}$$

3.2.1 Modelo YUV

O modelo de cor YUV é um modelo que tira proveito da fraca perceção da visão humana às variações de cor em relação às variações de luminosidade de uma imagem [27]. O valor de cor de um pixel é constituído por três componentes que correspondem à componente luma, o brilho de uma imagem, e duas outras componentes de crominância correspondentes às componentes de cor segundo uma projeção de cor azul e vermelha, respetivamente.

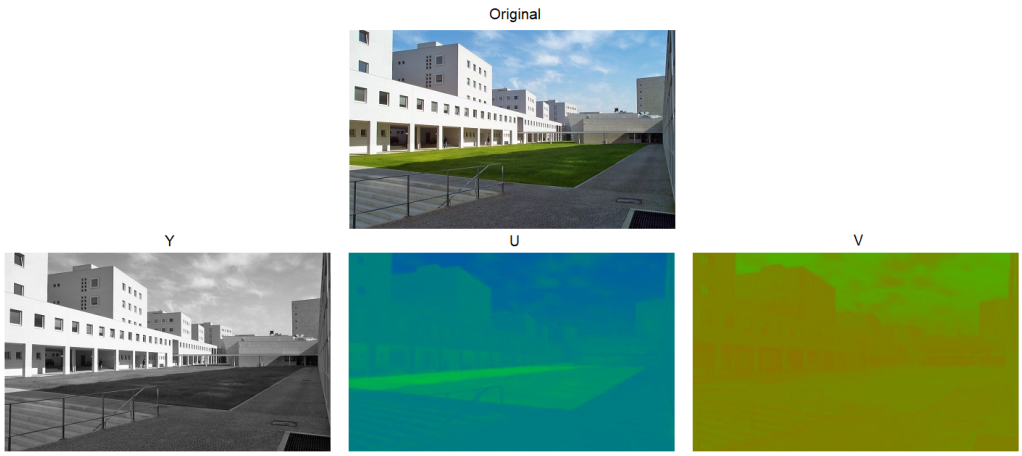

Figura 3.1: Decomposição de uma imagem nas componentes do modelo de cor YUV.

Segundo este modelo, a cor de um ponto de uma imagem é igual ao resultado da junção dos valores de intensidade de cada uma das três componentes referidas. Pode ser visualizado na

figura 3.1, a decomposição de uma imagem nas diferentes componentes de luma e crominância do modelo de cor YUV.

Devido à fraca perceção visual humana a alterações de cor de uma imagem, segundo este modelo de cor é possível reduzir o número de amostras que constituem as componentes de crominância e assim reduzir o tamanho das imagens. Esta prática é designada de subamostragem de crominâncias.

3.2.1.1 Subamostragem de Crominâncias

A subamostragem de crominâncias é a prática que consiste em reduzir o número de amostras das componentes de crominâncias de uma imagem segundo o modelo YUV. Esta prática permite reduzir o tamanho utilizado na representação de uma imagem através da partilha de valores das componentes de crominância por vários blocos de pixeis [7]. A subamostragem de crominâncias é definida por diferentes tipos que podem ser observados na seguinte figura 3.2:

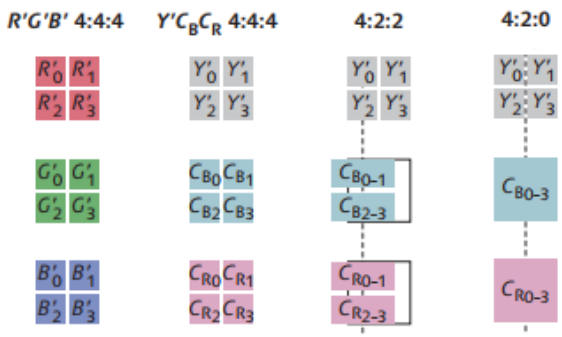

Figura 3.2: Diferentes tipos de subamostragem de crominâncias [7].

Segundo uma subamostragem do tipo 4:4:4, cada pixel é constituído por um valor próprio de cada uma das componentes de cor do modelo YUV. Com uma subamostragem do tipo 4:2:2, cada dois pixeis horizontais consecutivos partilham o mesmo valor das componentes de crominância, mantendo apenas cada um o seu respetivo valor de luma. Por último, com uma subamostragem do tipo 4:2:0, cada bloco de quatro pixeis adjacentes partilham os mesmos valores das componentes de crominância.

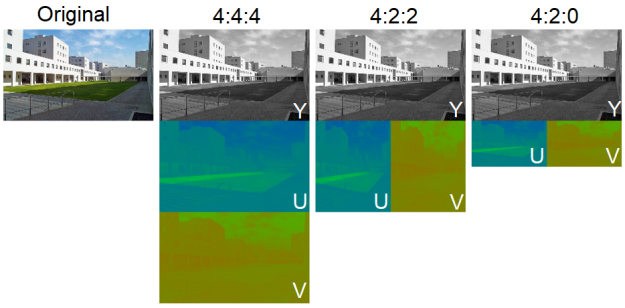

Figura 3.3: Subamostragem de crominâncias segundo o modelo YUV.

A figura 3.3 não apresenta os resultados finais da junção das componentes de cor por cada tipo de subamostragem, pois as diferenças entre as últimas e a imagem original não são percetíveis.

Este mecanismo de subamostragem de crominâncias é a principal razão da utilização frequente do modelo de cor YUV na área de multimédia para representar conteúdos de vídeo. Com a subamostragem de crominâncias é possível reduzir o tamanho de representação dos vídeos assim como diminuir o esforço computacional do seu processamento [25]. Entre os dois tipos de subamostragem 4:4:4 e 4:2:0, por exemplo, existe uma redução em 50% do número de amostras das componentes de crominância da imagem, o que por consequência indica uma diminuição do seu tamanho [28].

3.2.1.2 Profundidade de Cor

A profundidade de cor é o número de bits utilizados na representação do valor de cada componente de cor de um pixel. Quanto maior o valor da profundidade de cor maior será o intervalo de valores que cada componente pode representar e, por consequência, maior será a variedade de tonalidades de cor que um pixel pode tomar [28].

A figura 3.4 demonstra as diferenças das tonalidades de cor presentes numa imagem com diferentes valores de profundidade de cor, respetivamente: 8, 4, 2 e 1 bit(s).

Figura 3.4: Variação de tonalidades de cor em função do valor de profundidade de cor.

3.3 Reamostragem e Redimensionamento

A reamostragem e redimensionamento de vídeo é o processo de alteração das dimensões das frames que o constituem e é um processo realizado na maioria dos dispositivos de reprodução de vídeo. As dimensões de um vídeo são expandidas ou reduzidas de modo a que o último seja ajustado a uma determinada área de exibição ou para satisfazer uma restrição imposta de tamanho de imagem.

A reamostragem de sinal é o processo de alteração da taxa de amostragem de um sinal discreto baseado no sinal contínuo subjacente [29]. A alteração da taxa de amostragem de um sinal refere-se à mudança do número de amostras que o constitui. Em específico ao tema deste trabalho, a alteração da taxa de amostragem de um vídeo modifica o número de pixeis que constituem cada frame. A inserção de novos pixeis numa frame implica o aumento das suas dimensões, ou *upscaling*, enquanto a diminuição das suas dimensões, ou *downscaling*, implica a remoção de parte dos pixeis.

A reamostragem e redimensionamento de uma frame não resultam apenas da alteração do número de pixeis que a constitui mas, também, a partir da correção dos valores de intensidade de cor dos pixeis das frames reamostradas para que as caraterísticas visuais do vídeo sejam mantidas o quanto possível. Para realizar este processo podem ser utilizados uma variedade de algoritmos que balanceiam a qualidade dos resultados e o tempo de execução da aplicação do processo.

Como com qualquer algoritmo de reamostragem de sinal, a alteração das dimensões de um vídeo induz artefactos como desfocagem dos detalhes e inserção de ruído na imagem, assim como distorção das caraterísticas visuais apresentadas [30].

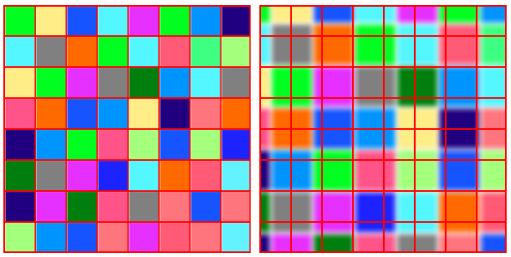

Figura 3.5: Desalinhamento entre pixeis reamostrados e a sua disposição original.

A dimensão com que as imagens são criadas têm em conta a aparência das suas características ao serem representadas numa determinada resolução. A alteração da dimensão de uma imagem provoca um desalinhamento entre a disposição original dos seus pixeis e as novas posições dos pixeis da imagem redimensionada. A imagem da figura 3.5 representa o desalinhamento consequente da reamostragem de uma imagem em relação à disposição original dos seus pixeis.

Considerando uma imagem original I_s de dimensões W_S de largura e H_s de altura, e uma imagem reamostrada I_t de dimensões W_t de largura e H_t de altura, os fatores de reamostragem horizontal e vertical são, respetivamente, R_h e R_v. Estes fatores, formulados pelas equações 3.7 e 3.8, são o valor da razão do número de amostras da imagem reamostrada e da imagem original.

$$R_h = \frac{W_t}{W_s} \tag{3.7}$$

$$R_v = \frac{H_t}{H_s} \tag{3.8}$$

Assumem-se as funções $I_s(x,y)$ e $I_t(x,y)$ como formulações do mapeamento do valor de intensidade de cor de um pixel na coluna x e na linha y da imagem original e da imagem reamostrada, respetivamente. Como as imagens I_s e I_t apresentam diferentes número de amostras, a posição de um pixel da imagem reamostrada $I_t(x,y)$ é calculada a partir da seguinte equação:

$$I_t(x,y) = I_s(\frac{x}{R_h}, \frac{y}{R_v}) \tag{3.9}$$

Por essa razão, a posição de um pixel da imagem reamostrada pode ser mapeada, dependendo do fator de amostragem, a uma posição contínua da imagem original I_s.

Pela definição de reamostragem, um certo sinal amostrado é reconstruido de modo a gerar um sinal contínuo de dados. O último é amostrado a uma taxa de amostragem diferente do sinal inicial. Desta forma, é possível calcular o valor de intensidade de cor de um pixel reamostrado que se encontra numa posição contínua da imagem original. Esta definição descreve o comportamento de interpolação [31].

Assim, a operação de reamostragem pode ser formulada como uma convolução que define o princípio geral de interpolação [32], apresentada em (3.10). As variáveis u e v correspondem à posição de um pixel da imagem reamostrada no sistema de coordenadas da imagem original e são dados a partir de $u = x/R_h$ e $v = y/R_v$, respetivamente.

O *kernel* de interpolação $k(x)$, também designado de filtro de reconstrução, é o método utilizado para interpolar os dados da imagem original de modo a reconstruir o sinal contínuo da mesma. A variável a é o valor de apoio de pixeis dos filtros de reconstrução e define o número de amostras originais a serem tidas em conta na operação de interpolação.

$$I_t(x,y) = \sum_{i=\lfloor u \rfloor - a + 1}^{\lfloor u \rfloor + a} \sum_{j=\lfloor v \rfloor - a + 1}^{\lfloor v \rfloor + a} I_s(\lfloor u \rfloor, \lfloor v \rfloor) \times k(u-i) \times k(v-j) \tag{3.10}$$

3.3.1 Filtros de Reconstrução

Os filtros de reconstrução são o aspeto fundamental das operações de interpolação, visto que os últimos apresentam diferentes complexidades de reconstrução do sinal contínuo a partir de amostras. Os filtros de reconstrução são caraterizados pelo seu valor de apoio de pixeis, o seu tempo de execução da interpolação e a qualidade dos resultados relativamente ao filtro de reconstrução ideal, o filtro *sinc*. O filtro *sinc* é um filtro do tipo IIR, ou *Infinite Impulse Response*, o que indica que este filtro tem uma resposta de impulso infinito [33]. A aplicação deste filtro na operação de reamostragem de imagens seria uma abordagem muito complexa, visto que o valor de intensidade de cor de cada pixel da imagem reamostrada seria calculado a partir da interpolação de todos os pixeis da imagem original [34].

A aplicação do filtro de reconstrução ideal *sinc* na operação de reamostragem e redimensionamento de imagens tem uma complexidade temporal de $(W_t \times H_t)^{W_s \times H_s}$. Porém, esta complexidade pode ser reduzida para $(W_t \times H_t)^{a^2}$ utilizando uma abordagem por regiões de processamento. Esta última abordagem refere-se ao processamento de um pixel da imagem reamostrada através de um filtro de reconstrução considerando apenas a amostras da cada dimensão da imagem original.

O filtro de reconstrução é o método utilizado para atribuir um peso ao valor de intensidade de cor de uma amostra da imagem original no cálculo de um pixel da imagem reamostrada. O peso calculado pelo filtro de reconstrução é obtido em função da distância entre as posições da amostra da imagem original e da posição do pixel da imagem reamostrada no sistema de coordenadas da imagem original.

3.3.1.1 *Box Filter*

O *box filter*, também designado de interpolação de *Nearest Neighbor*, é um filtro de interpolação com um apoio de pixel de $a = 2$, o que indica que durante a interpolação apenas quatro amostras originais são utilizadas para obter o valor de intensidade de cor do pixel da imagem reamostrada. Com este filtro, o valor de intensidade de cor de um pixel da imagem reamostrada é igual ao valor do pixel mais próximo da imagem original. Este tipo de interpolação tem um reduzido tempo de execução, visto que não é necessário cálculos aritméticos complexos mas apenas uma atribuição do valor da intensidade de cor da imagem original ao pixel da imagem reamostrada.

Figura 3.6: Qualidade dos resultados da operação de interpolação com o filtro *Nearest Neighbor*.

A qualidade dos resultados obtidos por esta interpolação apresentam uma acentuada ocorrência de *aliasing* [35]. Termo que designa as diferenças abruptas entre os gradientes de cor das caraterísticas visuais de uma imagem, como pode ser visualizado na figura 3.6. O filtro *Nearest Neighbor* é formulado matematicamente segundo a seguinte equação:

$$k_{nn}(x) = \begin{cases} 1 & |x| < 0.5 \\ 0 & \textit{senão} \end{cases} \tag{3.11}$$

3.3.1.2 Filtro Linear

O filtro de reconstrução linear, também designado de interpolação bilinear ou *Tent filter*, é um filtro de interpolação com valor de apoio de pixeis $a = 2$. Segundo este filtro, o valor de intensidade de cor de um pixel da imagem escalada será igual à média ponderada dos quatro pixeis originais mais próximos.

O filtro linear é uma interpolação mais sofisticada em relação ao filtro *Nearest Neighbor*, pois são utilizadas operações de interpolação linear para calcular o valor de intensidade de cor de um pixel da imagem reamostrada. Quanto maior for a proximidade de um pixel original ao pixel da imagem escalada, maior será a sua influência no valor de intensidade de cor calculada.

Figura 3.7: Qualidade dos resultados da operação de interpolação com o filtro linear.

Este filtro introduz novos valores de intensidade de cor que poderiam não existir na imagem original por consequência das interpolações lineares realizadas. Por esta razão, as imagens reamostradas recorrendo a este filtro de reconstrução apresentam características desfocadas especialmente em locais da imagem em que existem grandes discrepâncias de intensidade de cor, como pode ser observado na figura 3.7. O filtro de interpolação linear apresentado tem a seguinte formulação matemática:

$$k_{linear}(x) = \begin{cases} 1 - |x| & |x| < 1 \\ 0 & \textit{senão} \end{cases} \tag{3.12}$$

3.3.1.3 Spline Filter

Os resultados do filtro de reconstrução por *Spline* apresentam uma melhor qualidade de resultados comparativamente aos resultados dos filtros anteriores [35], pois o valor de intensidade de cor de cada pixel da imagem reamostrada é obtido pela interpolação cúbica dos dezasseis pixeis da imagem original mais próximos, de acordo com o valor de apoio de pixeis de $a = 4$. A boa qualidade dos resultados deste filtro é contrabalançada com o moroso tempo de execução da sua aplicação.

O filtro por *Spline* tem uma variedade de formulações matemáticas que determinam a presença e intensidade de artefactos na imagem reamostrada [36]. Os artefactos de imagem mencionados são a desfocagem e distorção das caraterísticas visuais, e erros no cálculo de valores interpolados em transições abruptas de gradiente, também designado de *ringing*.

$$k_{spline}(x) = \frac{1}{6}\begin{cases} (12-9\times B-6\times C)\times|x|^3+ & |x|<1 \\ \quad (-18+12\times B+6\times C)\times|x|^2+(6-2\times B) & \\ (-B-6\times C)\times|x|^3+(6\times B+30\times C)\times|x|^2+ & 1\le|x|<2 \\ \quad (-12\times B-48\times C)\times|x|+(8\times B+24\times C) & \\ 0 & \textit{senão} \end{cases} \tag{3.13}$$

A família de filtros de reconstrução por *Spline* é formulado segundo a equação (3.13), onde B e C regulam a qualidade dos resultados e a presença dos artefactos de imagem mencionados.

Figura 3.8: Qualidade dos resultados da operação de interpolação com o filtro por *Spline*.

Segundo Mitchell e Netravali, o filtro de reconstrução por *Spline* com o valor de B igual a 0 e C igual a 0.6, produz resultados próximos da qualidade de resultados que seriam esperados com o filtro de reconstrução ideal *sinc*, apesar da indução de artefactos de imagem *ringing* [36]. A formulação matemática do filtro de Mitchell e Netravali é apresentada pela seguinte equação:

$$k_{spline}(x) = \begin{cases} 1.4\times|x|^3-2.4\times|x|^2+1 & |x|<1 \\ -0.6\times|x|^3+3\times|x|^2-4.8\times|x|+2.4 & 1\le|x|<2 \\ 0 & \textit{senão} \end{cases} \tag{3.14}$$

3.3.1.4 Visão Geral

Na figura 3.9 são apresentados os resultados de cada um dos filtros de reconstrução onde é possível visualizar as diferenças entre as caraterísticas das imagens reamostradas e a presença dos diferentes artefactos de imagem. É possível verificar as transições abruptas de gradiente nas arestas dos detalhes da imagem pela interpolação *Nearest Neighbor*, a desfocagem das caraterísticas visuais do filtro linear e o aretegacto de *ringing* do filtro por *Spline* de Mitchell e Netravali.

Figura 3.9: Comparação de resultados entre os diferentes filtros de reconstrução.

3.4 Formatos de Pixeis

De modo a implementar uma solução de processamento de uma imagem, é necessário entender as diferentes peculiaridades de representação de um pixel na componente de memória de um computador. Na seguinte secção serão abordados os diferentes formatos de representação interna de pixeis do modelo de cor YUV. Os formatos de pixeis especificam o número de bits utilizados na representação de cada componente de cor de um pixel e a sua organização de acordo com o tipo de subamostragem de crominâncias da imagem.

3.4.1 Formatos Entrelaçados

Os formatos entrelaçados de pixeis são caraterizados pela representação consecutiva de pixeis. Os pixeis deste tipo de formatos são armazenados numa região contígua de memória. O formato de pixeis UYVY é uma das representações mais utilizadas entre os formatos entrelaçados. Segundo o formato UYVY, cada componente de cor é representada por um valor de profundidade de cor igual a 8 bits. Também, utiliza uma subamostragem de crominâncias do tipo 4:2:2, o que indica a partilha dos valores de crominância entre cada dois pixeis consecutivos.

Plano A

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
U0	U0	U0	U0	U0	U0	U0	U0	Y0	Y0	Y0	Y0	Y0	Y0	Y0	Y0	V0	V0	V0	V0	V0	V0	V0	V0	Y1	Y1	Y1	Y1	Y1	Y1	Y1	Y1

Figura 3.10: Representação interna do formato de pixel UYVY.

Na figura 3.10 é apresentada a representação interna do formato de pixel UYVY na componente de memória de uma máquina. Os formatos entrelaçados são caraterizados pela representação de todos os pixeis e as suas componentes de cor numa região contígua de memória, designada na figura como *Plano A*. Este formato segue o mesmo padrão de representação das componentes de cor de cada par de pixeis consecutivos: componente crominância U, seguida da componente luma do primeiro pixel do par, a componente de crominância V e, por último, o valor de luma do segundo pixel do par.

3.4.2 Formatos Planares

Os formatos planares de pixeis são caraterizados pela representação individual das componentes de cor de todos os pixeis de uma imagem. Cada componente de cor é alocada a uma região independente de memória. Os formatos de pixeis YUV422p e YUV420p são exemplos de formatos planares de pixeis. As diferentes componentes de cor, a componente de luma e as duas componentes de crominâncias, são organizadas cada uma de forma individual em relação às restantes.

Plano A

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Y0	Y0	Y0	Y0	Y0	Y0	Y0	Y0	Y1	Y1	Y1	Y1	Y1	Y1	Y1	Y1	Y2	Y2	Y2	Y2	Y2	Y2	Y2	Y2	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Y4	Y4	Y4	Y4	Y4	Y4	Y4	Y4	Y5	Y5	Y5	Y5	Y5	Y5	Y5	Y5	Y6	Y6	Y6	Y6	Y6	Y6	Y6	Y6	Y7	Y7	Y7	Y7	Y7	Y7	Y7	Y7

Plano B

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
U0	U0	U0	U0	U0	U0	U0	U0	U2	U2	U2	U2	U2	U2	U2	U2
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
U4	U4	U4	U4	U4	U4	U4	U4	U6	U6	U6	U6	U6	U6	U6	U6

Plano C

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V0	V0	V0	V0	V0	V0	V0	V0	V2	V2	V2	V2	V2	V2	V2	V2
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
V4	V4	V4	V4	V4	V4	V4	V4	V6	V6	V6	V6	V6	V6	V6	V6

Figura 3.11: Representação interna do formato de pixel YUV422p.

Na figura 3.11 é apresentada a representação em memória dos pixeis de uma imagem segundo o formato de pixeis YUV422p. Este tipo de formato é especifico a subamostragem de crominâncias do tipo 4:2:2 e, por essa razão, cada uma das componentes de crominância é constituída por metade do número de amostras da componente de luma. Também, este formato é distinto a componentes de cor com profundidade de 8 bits.

Nas figuras 3.11 e 3.12, o plano A corresponde à representação da componente de luma, o plano B e C à representação das componentes de crominância segundo uma projeção azul e vermelha, respetivamente, em regiões independentes de memória.

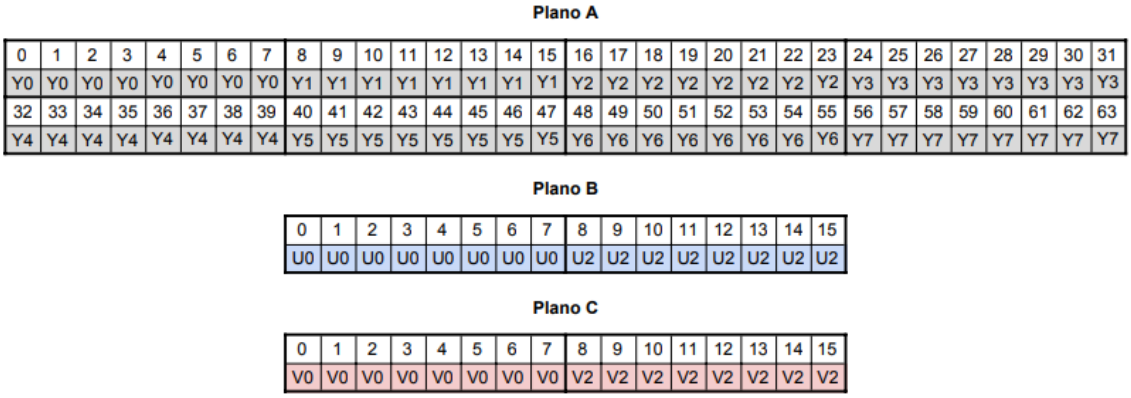

Figura 3.12: Representação interna do formato de pixel YUV420p.

O formato YUV420p apresenta as mesmas caraterísticas de representação de pixeis do formato YUV422p. Contudo, o formato YUV420p é destinado à representação de pixeis de imagem que seguem uma subamostragem de crominâncias do tipo 4:2:0, o que difere do formato YUV422p. Essa peculiaridade pode ser observado pelo menor número de amostras das componentes de crominância na representação do mesmo número de pixeis na figura 3.12.

3.4.3 Formatos Semi-Planares

Os formatos de pixeis semi-planares consideram parte das caraterísticas de representação de pixeis dos tipos de formatos mencionados nas secções anteriores. Isto é, os modelos semi-planares representam separadamente as componentes de crominâncias, à semelhança dos formatos planares de pixeis, e o entrelaçamento dos seus valores, como nos formatos de pixeis entrelaçados.

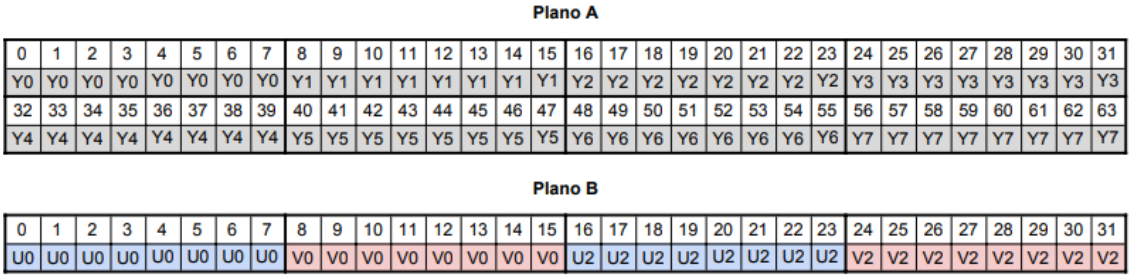

Figura 3.13: Representação interna do formato de pixel NV12.

O formato de pixeis NV12 é um formato semi-planar do modelo de cor YUV. Como pode ser observado na figura 3.13, à semelhança dos formatos planares, a componente de luma é representada de forma independente das restantes componentes de crominância. Contudo, a organização dos valores das componentes de crominâncias seguem as caraterísticas dos formatos entrelaçados. O formato NV12 é destinado à representação de pixeis de uma imagem cujo tipo de subamostragem de crominâncias é o tipo 4:2:0, com uma profundidade de cor de cada componente de 8 bits.

3.4.4 Outros Formatos

Os formatos de pixeis entrelaçados, planares e semi-planares detalham a maioria dos formatos utilizados segundo o modelo de cor YUV. Contudo, outros formatos especificam diferentes tipos de representação de pixeis de uma imagem.

O formato V210 especifica as caraterísticas de representação de pixeis de imagens em que cada valor de intensidade de cor de uma componente é representada com uma profundidade de 10 bits e segundo o tipo de subamostragem de crominâncias 4:2:2.

Devido à profundidade de cor de 10 bits deste formato e à arquitetura binária das máquinas, existe um desalinhamento entre a representação das componentes de cor de um pixel e o tipo de dados utilizado para a sua representação. Como se pode observar na figura 3.14, a representação de três valores de componentes de cor não utilizam totalmente o bloco de memória de 32 bits reservado ao efeito. A não utilização eficaz do espaço de memória reservado para a representação de pixeis de uma imagem é uma das peculiaridades deste tipo de formato.

Plano A

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
U0	U0	U0	U0	U0	U0	U0	U0	U0	U0	Y0	Y0	Y0	Y0	Y0	Y0	Y0	Y0	Y0	Y0	V0	V0	V0	V0	V0	V0	V0	V0	V0	V0	-	-
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Y1	Y1	Y1	Y1	Y1	Y1	Y1	Y1	Y1	Y1	U2	U2	U2	U2	U2	U2	U2	U2	U2	U2	Y2	Y2	Y2	Y2	Y2	Y2	Y2	Y2	Y2	Y2	-	-
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
V2	V2	V2	V2	V2	V2	V2	V2	V2	V2	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3	U4	U4	U4	U4	U4	U4	U4	U4	U4	U4	-	-
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
Y4	Y4	Y4	Y4	Y4	Y4	Y4	Y4	Y4	Y4	V4	V4	V4	V4	V4	V4	V4	V4	V4	V4	Y5	Y5	Y5	Y5	Y5	Y5	Y5	Y5	Y5	Y5	-	-

Figura 3.14: Representação interna do formato de pixel V210.

Na figura 3.14 é possível observar a organização dos valores das componentes de cor de uma imagem constituída por seis pixeis, segundo o formato de pixeis V210. É de notar a semelhança do padrão de representação das componentes com o formato entrelaçado UYVY, no sentido em que existe uma repetição do padrão de componentes de cor: componente crominância U, seguida da componente luma do primeiro pixel do par, a componente de crominância V e, por último, o valor de luma do segundo pixel do par.

3.5 Ferramentas de Processamento de Vídeo

A ferramenta FFmpeg é um projeto *open source* que fornece um conjunto de bibliotecas e programas para processamento de vídeo, áudio, *streams* e outros tipos de conteúdos multimédia. O FFmpeg é maioritariamente utilizado para transcodificação, codificação e descodificação de vídeo, edição de conteúdos multimédia, aplicação de operações de pós-produção de vídeo e para a introdução de conformidade nos conteúdos quanto aos standards utilizados pelos mesmos, ou também designado de standard *compliance*.

Várias entidades líderes de mercado da área de processamento gráfico como a Intel, NVidia e AMD fornecem circuitos integrados específicos para a realização de operações de codificação e

descodificação de vídeo. Porém, estes circuitos têm interfaces específicas e não apresentam características comuns entre si em termos de implementações e arquiteturas dos circuitos. O FFmpeg suporta como abstração as várias interfaces das entidades referidas e distribui de forma normalizada a capacidade da sua utilização.

A ferramenta FFmpeg permite a orquestração de todo um sistema de processamento e pós-produção de conteúdos multimédia a partir da própria implementação de protocolos de comunicação predominantemente utilizados por software de pós-produção de conteúdos multimédia. Em conjunto com o extenso suporte do FFmpeg aos principais *muxers*, *codecs*, modelos de cor e respetivos formatos de pixeis, o FFmpeg é uma das bibliotecas mais utilizadas na manipulação de conteúdos multimédia, pois encontra-se na base de software como o VLC Media Player, Blender, Plex, Youtube, Google Chrome, Mozilla Firefox, entre outros.

Os *muxers* são modelos que determinam a forma de representação de todos os dados e metadados de diferentes *codecs* de vídeo e áudio numa só especificação. Os *codecs* definem as propriedades de representação dos dados de um conteúdo multimédia, pormenorizam a codificação e descodificação de sinais que podem ser realizados em hardware ou em software. Os conteúdos multimédia são definidos pelo tipo de *codec* que utilizam, pois o último indica a organização e representação do seu sinal [37].

Software como o Adobe Premiere, Sony Vegas, MainConcept e Elecard são ferramentas bastante utilizadas para a aplicação de processos de pós-produção. Contudo, a sua utilização está dependente de licenciamento o que torna inviável a sua utilização como referência no trabalho desta dissertação.

As ferramentas e bibliotecas *open source* de processamento de imagem, como o ImageMagick ou OpenCV, não integram nativamente o modelo de cor YUV e, por essa razão, não suportam nativamente operações de processamento de vídeo. Assim estas ferramentas foram desconsideradas no desenvolvimento deste trabalho, pois a sua utilização implicaria um nível de implementação adicional já presente com a ferramenta FFmpeg.

3.6 Conclusão

Como já referido no capítulo 1, o objetivo prático deste trabalho é a otimização do processo de reamostragem e redimensionamento de vídeo sem compressão. Devido à caraterística digital dos vídeos, o modelo de cor YUV é o mais apropriado para a sua representação e processamento. Por essa razão, apenas este modelo de cor foi analisado e utilizado neste trabalho.

Os filtros de reconstrução *Nearest Neighbor*, linear e por *spline* são os filtros que expõem os conceitos gerais e indicam o comportamento esperado dos resultados do processo de reamostragem e redimensionamento de vídeo. Contudo, existem outros filtros de reconstrução como os filtros *Lanczos*, *Catmull-Rom* e *Sinc2* que não foram abordados. Os últimos são semelhantes aos filtros apresentados na secção 3.3.1, diferindo apenas em pequenos detalhes de formulação matemática. Por essa razão, não foram considerados relevantes para o desenvolvimento deste trabalho.

De igual forma, os formatos de pixeis apresentados na secção 3.4 não cobrem todos os formatos de pixeis do modelo de cor YUV. Contudo, apenas foram mencionados os formatos referidos pois mostram de forma geral as caraterísticas de representação de cada tipo de formato de pixel mais comum na área de pós-produção de conteúdos multimédia.

O tema desta dissertação foi proposta por uma empresa da área de multimédia que utiliza a ferramenta FFmpeg como o principal recurso de automatização do seu produto de pós-produção de conteúdos multimédia. Como o resultado deste trabalho irá substituir, idealmente, a implementação atual do processo de reamostragem e redimensionamento de vídeo do produto da empresa, a comparação mais correta deve ser realizada entre a ferramenta FFmpeg e a solução implementada neste trabalho.

Capítulo 4

Implementação Heterogénea de Reamostragem de Vídeo

Neste capítulo é descrita e detalhada a solução proposta para o problema de aplicação do processo de reamostragem e redimensionamento de vídeo sem compressão, segundo o modelo de cor YUV, com o objetivo de reduzir o tempo de execução. Na descrição da solução proposta são abordadas as fases que constituem a mesma e as respetivas otimizações implementadas. Por último, são discutidos os detalhes e as decisões de implementação tomadas, assim como as respetivas justificações.

4.1 Descrição

As imagens do modelo de cor YUV são caraterizadas por múltiplos formatos de pixeis, vários tipos de subamostragem de crominâncias e diferentes valores de profundidade de cor, como já referido na secção 3.4. Estes diferentes aspetos definem múltiplas combinações de caraterísticas de representação de imagens. Por esta razão, a solução deste trabalho deve considerar uma implementação que lida com as várias combinações de caraterísticas possíveis.

De modo a normalizar a lógica da solução deste trabalho foi apenas considerada a aplicação do processo de reamostragem e redimensionamento de imagens segundo o formato de pixeis planares. Caso a imagem a ser processada não apresente este tipo de formato, a última será convertida para um formato planar de pixeis que assegure os mesmos valores de profundidade de cor e o tipo de subamostragem de crominâncias quer da imagem original quer da imagem reamostrada. Então, a solução é dividida em duas fases distintas: a fase de conversão de formatos de pixeis de frames e a fase de reamostragem e redimensionamento das imagens.

A lógica da solução implementada pode ser observada pelo diagrama de atividade da figura 4.1. Quando o processamento da operação de reamostragem e redimensionamento é aplicado a

uma imagem sem indicação de alteração das suas dimensões, a última sofre apenas a operação da fase de conversão de formatos de pixeis: do formato inicial para o formato final desejado.

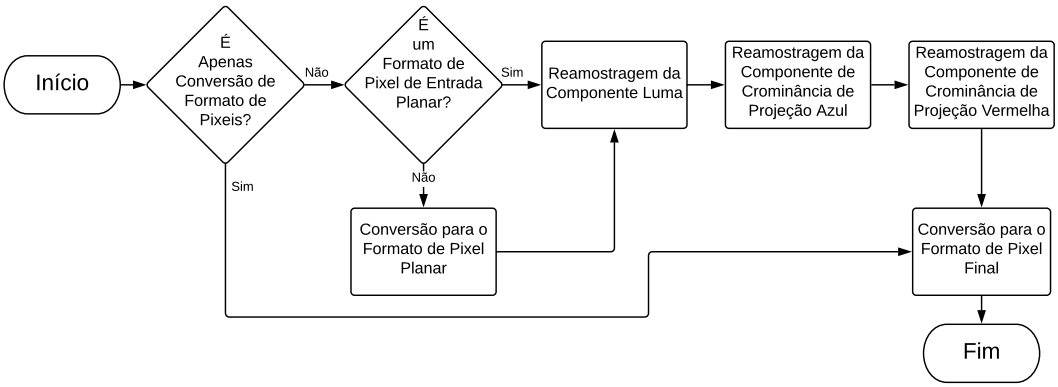

Figura 4.1: Diagrama de atividade da solução para o processamento de uma frame.

No caso em que exista a indicação da alteração das dimensões da imagem, o processamento é constituído pela sequência das seguintes três operações: a operação de conversão do formato de pixeis, que converte do formato de pixel da imagem original para um formato do tipo planar; de seguida, cada componente de cor (plano) da imagem é reamostrada e redimensionada individualmente recorrendo aos filtros de reconstrução; por último, a imagem reamostrada e redimensionada sofre novamente a operação de conversão do tipo de formato de pixeis, do formato planar temporário para o formato final desejado.

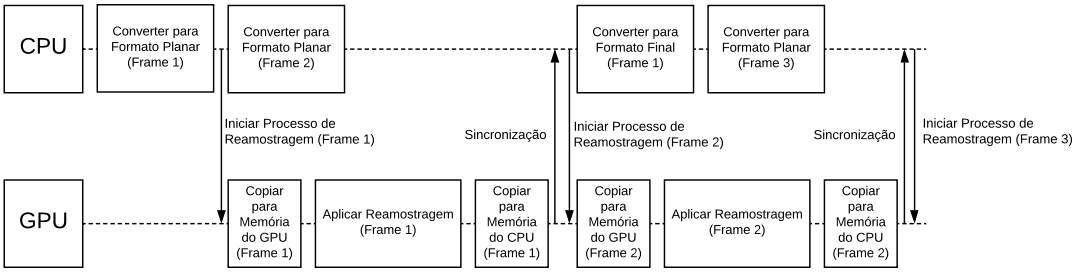

Figura 4.2: Diagrama de organização de tarefas de processamento das frames de um vídeo.

A figura 4.2 representa de forma esquemática a organização da atividade da solução desenvolvida no processamento das primeiras três frames de um vídeo, sendo que o restante processamento é uma replicação do padrão de operações apresentado (ver também o apêndice A.1).

De modo a tirar o máximo partido das capacidades computacionais do CPU e do GPU, ambas unidades de processamento executam diferentes operações em simultâneo, havendo apenas sincronização de ambas execuções entre operações de reamostragem e redimensionamento de frames de modo a que o processamento realizado não seja descoordenado e ineficiente. A solução foi implementada de forma a que a operação de conversão de formato de pixeis fosse realizada apenas pelo CPU e a operação de reamostragem e redimensionamento pelo GPU.

Considerando uma frame F_n de índice n de um vídeo, as duas unidades de processamento executam as fases da solução, referidas acima, da seguinte forma: enquanto a operação de reamostragem e redimensionamento da frame F_n é levada a cabo pelo GPU, o CPU realiza as operações de conversão do tipo de formatos de pixeis do formato planar intermédio para o formato final e do formato de pixeis inicial para o formato planar intermédio das frames F_{n-1} e F_{n+1}, respetivamente.

Esta divisão de processamento em fases permite uma utilização dos recursos computacionais das unidades de processamento próxima do valor máximo. A figura 4.3 apresenta a percentagem do tempo de execução de cada uma das fases referidas no processamento de um vídeo, segundo a implementação desta solução.

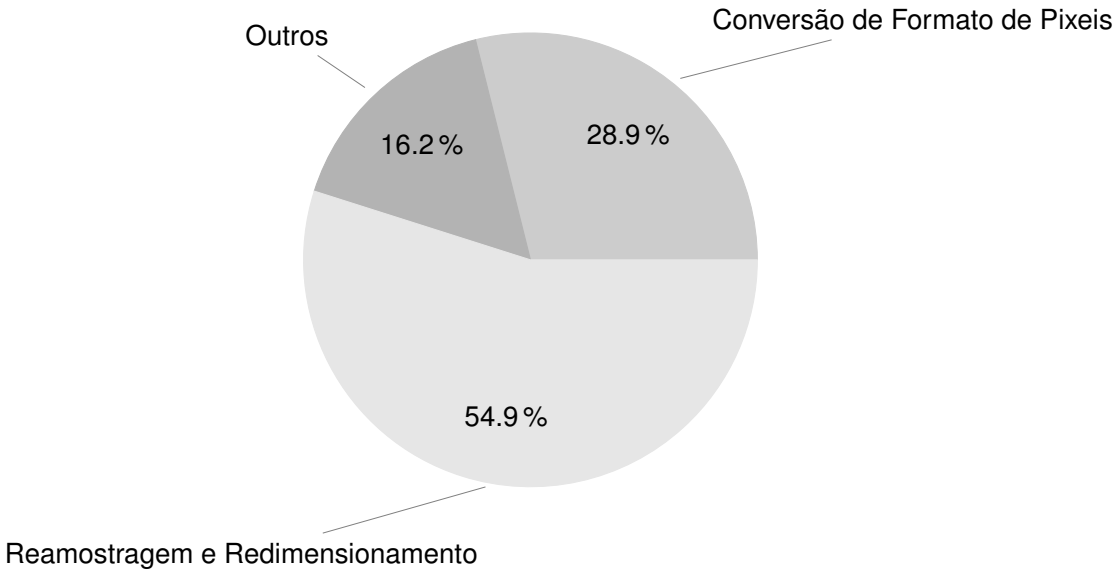

Figura 4.3: Distribuição de trabalho das fases de processamento.

A fase de reamostragem e redimensionamento de uma frame, executada no GPU, é o fator limitante de toda esta operação visto que é a operação mais morosa de toda a execução. Por essa razão, a utilização do CPU não é máxima, pois a fase de conversão do tipo de formatos de pixeis das frames F_{n-1} e F_{n+1} apresenta um tempo de execução menor que a fase de reamostragem e redimensionamento da frame F_n. Este acontecimento leva a que o CPU tenha períodos de inatividade enquanto aguarda os seguintes dados a serem processado e, por consequência, a utilização das suas capacidade computacionais não são levadas ao máximo.

De modo a evitar esta peculiaridade e maximizar a utilização do CPU, o último poderia continuar a sua execução e processar as restantes frames do vídeo, ao invés de bloquear e esperar pelo término da fase do GPU. Porém, este tipo de implementação é impossível visto que a solução tem como objetivo a aplicação do processo de pós-produção - reamostragem e redimensionamento - de vídeo *live*, que se encontra em captura e, por essa razão, o processamento das frames seguintes pode não ser possível, pois podem ainda não terem sido capturadas.

4.2 Operação de Conversão de Formatos de Pixeis

A operação de conversão do tipo de formatos de pixeis de uma imagem é realizada exclusivamente recorrendo às capacidades computacionais do CPU utilizando a ferramenta OpenMP para a sua implementação. A operação de conversão do formato de pixeis de uma imagem é fundamentalmente um problema de reorganização dos valores das componentes de cor dos seus pixeis. Por essa razão, não se justifica a utilização das capacidades computacionais do GPU, visto que a última está, exclusivamente, otimizada para a execução de problemas de cariz aritmético [21].

As limitações da aplicação desta operação devem-se ao elevado número de acessos a memória necessários de serem realizados para efetuar as operações de leitura e escrita dos valores de intensidade de cor de cada componente de um pixel. Assim, a chave para a otimização deste problema encontra-se na utilização de técnicas que implementem eficientemente os acessos a memória.

Os acessos aleatórios a memória são custosos devido à latência de transferência dos dados armazenados na memória para os registos do processador, caso os dados não se encontrem já armazenados no sistema de cache, os designados *page faults* [38]. Para minimizar a ocorrência de *page faults* é necessário ter em conta o comportamento do sistema de memória cache. Isto é, o número de ocorrências de *page faults* é reduzido caso os acessos a memória sejam feitos a regiões próximas de acessos realizados anteriormente de modo a ter uma maior probabilidade de encontrar os dados no sistema de cache [39].

Tendo em conta que a realização da operação de conversão do tipo de formatos pixeis de uma frame é executada por vários *threads*, implementados com a ferramenta OpenMP, é necessário assegurar uma divisão equilibrada dos recursos computacionais para efetuar a operação atendendo a um acesso eficiente a memória.

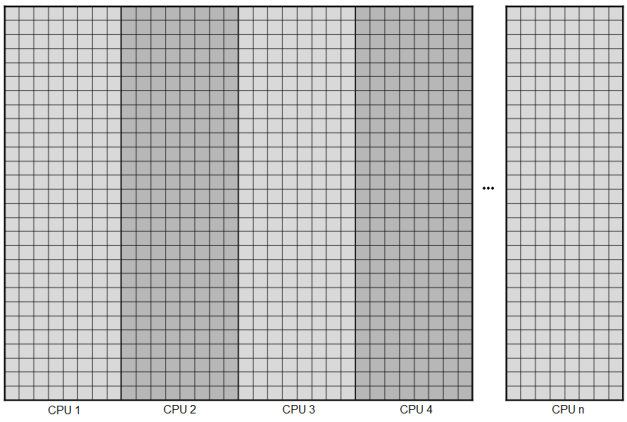

Figura 4.4: Divisão do processamento de uma frame em regiões.

A figura 4.4 estrutura a divisão de uma frame de um vídeo em regiões a serem processadas por diferentes *threads*. Cada quadrícula representa os pixeis de uma imagem e cada região sombreada os dados de uma imagem a serem atribuídos a cada *thread* dos *n* cores do CPU. Esta divisão permite que os acessos a memória conduzidos por cada um dos *threads* ocorra em regiões de memória

próximas das acedidas pelos restantes *threads*, tirando maior partido do sistema de memória cache e melhorando a eficiência de acessos a memória, devido à redução de *page faults*.

O seguinte excerto de código exemplifica a implementação da operação de conversão do tipo de formato de pixeis com a ferramenta OpenMP praticada na solução desenvolvida. O ciclo *for* exterior itera o processamento de todas as linhas da imagem, enquanto o ciclo interior itera todas as colunas da mesma.

A diretiva *pragma omp parallel* declara um bloco de código paralelo a ser executado em múltiplos *threads* do CPU como uma tarefa, enquanto a diretiva *pragma omp for schedule(static)*, imediatamente antes do ciclo *for* interior, dá a indicação ao CPU de paralelização do ciclo interior por todos os *threads* da secção paralelizada.

A carga de trabalho do ciclo será dividida por todos os *threads* de acordo com um certo padrão especificado pela palavra-chave *schedule(static)*. Esta diretiva dá indicação ao CPU que a divisão dos dados será realizada estaticamente durante a compilação de modo a que cada *thread* seja responsável pelo processamento de blocos de dados de dimensão $\frac{IMAGE\_WIDTH}{THREAD\_COUNT}$, sendo a variável $IMAGE\_WIDTH$ o valor da dimensão de largura da imagem e a variável $THREAD\_COUNT$ o número de threads que constituem a secção de código paralelizada.

```
1  #pragma omp parallel
2  {
3      // Itera por todas as linhas da imagem
4      for(int vIndex = 0; vIndex < IMAGE_HEIGHT; vIndex++){
5          // Paraleliza o processamento das colunas da imagem
6          #pragma omp for schedule(static)
7          for(int hIndex = 0; hIndex < IMAGE_WIDTH; hIndex++){
8              // Converte o formato de pixel do bloco de pixeis da linha vIndex
9              convertPixelFormatOfImageBlock(vIndex, hIndex);
10         }
11     }
12 }
```

Nos casos em que a operação de reamostragem e redimensionamento indica alterações das dimensões da imagem, a operação de conversão do tipo de formatos de pixeis do formato original para o formato planar intermédio é realizado de modo a reduzir o número de operações lógicas do processo de reamostragem. A redução do número de operações deve-se à organização dos dados dispostos por planos ao invés de necessitar de operações adicionais para separar os valores de componentes de cor entrelaçados.

Adicionalmente, a quantidade de dados a serem processados na operação de reamostragem e redimensionamento são reduzidos pela conversão de formato de pixeis, pois o formato planar intermédio terá o mesmo tipo de subamostragem de crominâncias que o tipo de subamostragem com menos valores de componentes de crominâncias da imagem original ou da imagem reamostrada. Assim, como exemplo, o processo de reamostragem e redimensionamento de um vídeo, caraterizado pelo tipo de subamostragem de crominâncias 4:2:2 para uma subamostragem do tipo

4:2:0, irá ser realizado com um formato planar intermédio com o tipo de subamostragem de crominâncias 4:2:0, reduzindo efetivamente o número de valores de intensidade de componentes de crominâncias a serem processados em 25%.

A tabela 4.1 mostra o tipo de subamostragem de crominâncias do formato de pixeis planar intermédio pelo qual é realizada a operação de reamostragem e redimensionamento, dependendo do tipo de formato de pixeis de entrada, da imagem original, e de saída, da imagem reamostrada.

Tabela 4.1: Tipo de subamostragem de crominâncias do formato de pixeis planar intermédio.

	Formato de Pixeis da Imagem Reamostrada				
Formato de Pixeis da Imagem Original	UYVY	YUV422p	YUV420p	NV12	V210
UYVY	4:2:2	4:2:2	4:2:0	4:2:0	4:2:2
YUV422p	4:2:2	4:2:2	4:2:0	4:2:0	4:2:2
YUV420p	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0
NV12	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0
V210	4:2:2	4:2:2	4:2:0	4:2:0	4:2:2

4.3 Operação de Reamostragem e Redimensionamento

A operação de reamostragem e redimensionamento é realizada, exclusivamente, recorrendo às capacidades computacionais do GPU, utilizando a plataforma de desenvolvimento CUDA para a sua implementação. Como referido na secção 3.3.1, a operação de reamostragem e redimensionamento de uma imagem é fundamentalmente a aplicação do cálculo de interpolação recorrendo aos filtros de reconstrução. Esta operação é conseguida pela aplicação de cálculos aritméticos sobre os dados das componentes de cor de uma imagem e, por essa razão, a natureza desta operação torna a utilização do GPU extremamente eficaz para a resolução deste problema [22].

A alocação de memória de uma unidade de processamento gráfico é uma operação síncrona com o CPU, bloqueando o último até o término dessa atividade. Como as caraterísticas da operação de reamostragem e redimensionamento de um vídeo são semelhantes para todas as suas frames e tendo em conta que durante esta fase apenas uma frame é processada de cada vez, o processo de alocação de memória pode ser realizado apenas uma vez no início de toda esta operação e libertada no seu final. Para esta operação é necessário reservar uma região de memória de dimensões igual às componentes de cor de uma frame do vídeo original e do vídeo reamostrado. A cada iteração do processamento de um vídeo, a memória do GPU recebe os dados das componentes de cor de uma frame sobrepondo os valores da frame anterior.

As limitações da aplicação desta operação são introduzidas pelas operações inevitáveis de transferência de dados das componentes de cor das frames entre o dispositivo de memória da máquina e da unidade de processamento gráfico, e o elevado tempo de resposta de acessos de memória. Estas limitações são provocadas, respetivamente, devido ao GPU apenas conseguir operar sobre dados que se encontrem em algum dos seus níveis de memória e à natureza de acessos não coalescidos a memória desta operação, como será explicado de seguida.

Uma das caraterísticas dos filtros de reconstrução utilizados na operação de reamostragem e redimensionamento é o seu valor de apoio de pixeis. O valor de apoio de pixeis indica o número de pixeis da imagem original mais próximos da posição do pixel da imagem reamostrada que são tidos em conta no cálculo do seu valor de intensidade de cor.

O valor de intensidade de cor de cada pixel da imagem reamostrada é obtido a partir do cálculo de interpolação por cada *thread* do GPU. Cada um dos *threads* necessita de realizar vários acessos a memória para recuperar os valores de intensidade de cor dos pixeis originais.

Assim, um *thread* não acede de forma coalescida a memória do GPU. Porém, como cada *thread* acede aos valores de intensidade de cor dos pixeis da imagem original mais próximos da posição do pixel da imagem reamostrada, é possível tirar partido do sistema de cache da memória de textura do GPU, por priorizar a localidade espacial dos dados (a proximidade das posições acedidas da memória) e assim reduzir a latência de resposta de acessos de memória realizados por cada *thread* [17], como descrito na secção 2.3.1.4.

4.3.1 Mecanismos de Transferência de Dados do GPU

A plataforma de desenvolvimento CUDA fornece diferentes mecanismos de transferência que permitem a cópia de dados a partir da memória da máquina para os níveis de memória das unidades de processamento gráfico. Esses métodos são os seguintes:

- Cópia de memória sincronizada - Este mecanismo permite reservar uma região de um dos níveis de memória do GPU, onde serão armazenados os dados transferidos da memória da máquina. Esta transferência é realizada sincronamente com o CPU, bloqueando-o até o término da atividade.

- Memória *pinned* - O mecanismo de memória *pinned* é utilizado pelo GPU para executar operações sobre dados sem que haja uma cópia explicita dos mesmos da memória da máquina para a memória da unidade de processamento gráfico. Este mecanismo é realizado através do procedimento DMA do GPU, ou *direct memory access*. O último é um procedimento que permite o acesso a memória sem que haja bloqueio de atividade da unidade de processamento que o invocou. O mecanismo de memória *pinned* permite a execução direta do GPU sobre dados armazenados na memória da máquina.

- Cópia de memória não paginada - Este tipo de transferência de dados é conseguido através da declaração como não paginável de uma região de memória da máquina. Esta declaração indica ao sistema operativo que a região não deve ser movida ou copiada para um ficheiro de paginação de memória. Ao declarar uma região de memória da máquina como não paginável, o GPU consegue realizar uma cópia dos seus dados assincronamente, não bloqueando a atividade quer do CPU quer da unidade de processamento gráfico.

Dos três mecanismos de transferência de dados entre a memória da máquina e do GPU, o mecanismo de memória não paginada é o que apresenta melhor desempenho e escalabilidade para

soluções de processamento paralelo de dados [40], pelo que foi o mecanismo implementado na solução deste trabalho.

4.3.2 Divisão da Carga Computacional

A plataforma de desenvolvimento CUDA estabelece o conceito de sequência de operações com a declaração de uma classe designada de *stream*. As instâncias desta classe permitem a paralelização de diferentes tarefas como a transferência de dados para a memória do GPU a partir de memória não paginável, a operação reversa e a execução de uma operação de processamento sobre os dados. Uma unidade de processamento gráfico tem a capacidade de executar as referidas tarefas em simultâneo desde que as últimas sejam de diferentes tipos e pertencentes a instâncias diferentes da classe *stream*.

Figura 4.5: Divisão de tarefas por *streams* no GPU.

A figura 4.5 reflete a divisão dos diferentes tipos de tarefas realizadas concorrentemente pelo GPU durante a operação de reamostragem e redimensionamento desta solução. Na figura, cada linha carateriza uma instância da classe *stream*, os blocos coloridos *Y*, *U* e *V* representam as operações de reamostragem e redimensionamento das três componentes de cor de uma frame, e os blocos *A1*, *A2*, *A3*, *B1*, *B2* e *B3* representam as transferências de dados entre o dispositivo de memória da máquina e o GPU.

Com a utilização de instâncias da classe *stream* foi permitido esconder a latência causada pelas transferências de dados de *A2*, *A3*, *B1* e *B2*, sobrepondo essas atividades com a execução das operações de reamostragem e redimensionamento.

O excerto de código 4.1 implementa a divisão de dados por tarefas e a sobreposição da sua execução, como encontrado na figura 4.5. Nesta implementação cada componente de cor de cada frame de um vídeo é processada separadamente. As instâncias da classe *stream* são criadas tendo em vista o processamento da operação de reamostragem concorrente das diferentes componentes de cor. Esta implementação permite que as transferências de dados das componentes de cor sejam realizadas assincronamente, o que permite a continuação de execução do CPU e do GPU.

```
1  // Aloca memoria na memoria de textura do GPU
2  cudaArray* Y_COMPONENT_GPU_ARRAY, * U_COMPONENT_GPU_ARRAY, * V_COMPONENT_GPU_ARRAY;
3  cudaMallocArray(&Y_COMPONENT_GPU_ARRAY, &TRANSFER_CHANNEL, WIDTH, HEIGHT);
4  cudaMallocArray(&U_COMPONENT_GPU_ARRAY, &TRANSFER_CHANNEL, CHROMA_WIDTH,
       CHROMA_HEIGHT);
5  cudaMallocArray(&V_COMPONENT_GPU_ARRAY, &TRANSFER_CHANNEL, CHROMA_WIDTH,
       CHROMA_HEIGHT);
6
7  // Aloca memoria no GPU atraves do mecanismo de memoria nao paginavel
8  uint8_t* Y_RESAMPLED_GPU_ARRAY, * U_RESAMPLED_GPU_ARRAY, * V_RESAMPLED_GPU_ARRAY;
9  cudaMallocHost((void **) Y_RESAMPLED_GPU_ARRAY, Y_RESAMPLED_COMPONENT_SIZE);
10 cudaMallocHost((void **) U_RESAMPLED_GPU_ARRAY, U_RESAMPLED_COMPONENT_SIZE);
11 cudaMallocHost((void **) V_RESAMPLED_GPU_ARRAY, V_RESAMPLED_COMPONENT_SIZE);
12
13 // Instanciacao das diferentes streams
14 cudaStream_t streamY, streamU, streamV;
15 cudaStreamCreate(&streamY);
16 cudaStreamCreate(&streamU);
17 cudaStreamCreate(&streamV);
18
19 // Copia assincronamente os dados para a memoria de textura do GPU
20 cudaMemcpyToArrayAsync(Y_COMPONENT_GPU_ARRAY, 0, 0, Y_COMPONENT_CPU_ARRAY,
       Y_COMPONENT_SIZE, cudaMemcpyHostToDevice, streamY);
21 cudaMemcpyToArrayAsync(U_COMPONENT_GPU_ARRAY, 0, 0, U_COMPONENT_CPU_ARRAY,
       U_COMPONENT_SIZE, cudaMemcpyHostToDevice, streamU);
22 cudaMemcpyToArrayAsync(V_COMPONENT_GPU_ARRAY, 0, 0, V_COMPONENT_CPU_ARRAY,
       V_COMPONENT_SIZE, cudaMemcpyHostToDevice, streamV);
23
24 // Operacao de reamostragem e redimensionamento das componentes de cor
25 resampleOperation<<<KERNEL_lAUNCH_PARAMETERS, streamY>>>(SOURCE_WIDTH,
       SOURCE_HEIGHT, TARGET_WIDTH, TARGET_HEIGHT, Y_COMPONENT_GPU_ARRAY,
       Y_RESAMPLED_GPU_ARRAY);
26 resampleOperation<<<KERNEL_lAUNCH_PARAMETERS, streamU>>>(SOURCE_WIDTH,
       SOURCE_HEIGHT, TARGET_WIDTH, TARGET_HEIGHT, U_COMPONENT_GPU_ARRAY,
       U_RESAMPLED_GPU_ARRAY);
27 resampleOperation<<<KERNEL_lAUNCH_PARAMETERS, streamV>>>(SOURCE_WIDTH,
       SOURCE_HEIGHT, TARGET_WIDTH, TARGET_HEIGHT, V_COMPONENT_GPU_ARRAY,
       V_RESAMPLED_GPU_ARRAY);
28
29 // Copia assincronamente os dados da memoria global do GPU
30 cudaMemcpyAsync(Y_COMPONENT_CPU_ARRAY, Y_RESAMPLED_GPU_ARRAY,
       Y_RESAMPLED_COMPONENT_SIZE, cudaMemcpyDeviceToHost, streamY);
31 cudaMemcpyAsync(U_COMPONENT_CPU_ARRAY, U_RESAMPLED_GPU_ARRAY,
       U_RESAMPLED_COMPONENT_SIZE, cudaMemcpyDeviceToHost, streamU);
32 cudaMemcpyAsync(V_COMPONENT_CPU_ARRAY, V_RESAMPLED_GPU_ARRAY,
       V_RESAMPLED_COMPONENT_SIZE, cudaMemcpyDeviceToHost, streamV);
```

Listing 4.1: Código implementado para a divisão da carga computacional das tarefas do GPU.

4.4 Comparação de *float* e *double*

A utilização das unidades de processamento gráfico na execução da operação de reamostragem e redimensionamento de frames de um vídeo restringe a precisão dos tipos de dados das variáveis utilizadas. Os GPUs têm uma maior afinidade com a utilização do tipo *float* de dados relativamente ao tipo *double*. Os GPUs manifestam um melhor tempo de execução de processamento de cálculos constituídos por dados do tipo *float*, ou de precisão singular, em comparação com o tempo de execução de cálculos constituídos por dados do tipo *double*, ou de precisão dupla [41].

Esta discrepância deve-se à arquitetura das unidades de processamento gráfico, pois estas unidades são constituídas por um maior número de componentes de processamento de operações de vírgula flutuante de precisão singular do que de precisão dupla [3].

Nesta secção é abordada a análise numérica da precisão associada à operação de reamostragem e redimensionamento utilizando variáveis do tipo *float* e do tipo *double*, comparando o impacto de cada um desses tipos na qualidade da imagem resultante.

4.4.1 Estrutura de Números de Vírgula Flutuante

Um número real é representado pela arquitetura de uma máquina segundo o standard IEEE 754, que especifica a arquitetura e aritmética computacional associada aos números de vírgula flutuante [42]. Segundo este standard, os tipos de dados *float* e *double* são representados pelos campos de sinal, expoente e mantissa.

Tabela 4.2: Tamanho em bits dos campos dos tipos de dados *float* e *double*.

	Tamanho em bits	
Campo	Float	Double
Sinal	1	1
Expoente	8	11
Mantissa	23	52

A tabela anterior mostra o tamanho em bits dos campos dos tipos de dados *float* e *double* para a representação de números de vírgula flutuante, segundo o standard IEEE 754. O número de bits dedicados ao expoente determina o alcance de grandeza do valor representado pelo tipo de dados da variável, enquanto o número de bits da mantissa determina a precisão da representação do seu valor.

Considerando um tipo de dados de vírgula flutuante com E bits reservados à representação do expoente, o alcance de grandeza do seu valor é dado por $[-2^{E-1}+1, 2^{E-1}] : E \in \mathbb{N}$. Como se pode induzir pela formulação matemática do alcance de grandeza do expoente, um dos bits do expoente não é utilizado para a representação do valor de expoente de um número real, pois segundo o standard IEEE 754 esse bit é utilizado no tratamento de exceções como o número zero e o número NAN, ou *not a number*, que representa valores de vírgula flutuante não representáveis ou não definidos [42].

4.4.2 Erro de Arredondamento Associado ao Tipo de Dados

Devido à limitação do número de bits reservados à representação da mantissa na arquitetura de uma máquina, existe um arredondamento do número real ao número de vírgula flutuante mais próximo, representável como *float* ou *double*.

Como exemplo, o número $(0.1)_{10}$ tem uma representação infinita em binário: $(1100(1100))_2$. De modo a ser possível representar este número como um número de vírgula flutuante, segundo o standard IEEE 754, é necessário restringi-lo ao número de bits da mantissa através de uma operação de arredondamento. O valor do arredondamento de um número real ao número de vírgula flutuante mais próximo representável pode ser entendido como o erro associado a essa conversão.

A função ULP, ou *Unit of Least Precision*, de um número real é o valor do bit menos significativo da mantissa na sua representação como um número de vírgula flutuante do standard IEEE 754. O valor da função ULP é o valor da diferença entre dois números de vírgula flutuante consecutivos. A função ULP tem um comportamento proporcionalmente direto à magnitude do valor do número real considerado [43].

Considera-se um número real x e os números de vírgula flutuante representáveis x_a e x_b mais próximos de x, de tal modo que $x_a < x < x_b$. Pela definição da função ULP, $x_b - x_a = ULP(x)$. Assim, se $\frac{ULP(x)}{2} < x_b - x$, então $x - x_a < \frac{ULP(x)}{2}$. Logo, é possível assumir que o erro máximo causado pelo arredondamento de um número real ao número de vírgula flutuante representável mais próximo, dos tipos especificados pelo standard IEEE 754, é no máximo $\frac{ULP(x)}{2}$.

De modo a generalizar o erro de arredondamento associado à representação de números reais como números de vírgula flutuante dos tipos *float* e *double*, considera-se um tipo de representação genérica com E e P bits reservados, respetivamente, ao expoente e à mantissa. O erro associado ao arredondamento de um número real x é formulado através da seguinte equação:

$$Erro(x) = \frac{ULP(x)}{2} = \frac{max(2^{-149}, 2^{\lfloor log_2(|x|)-P+1 \rfloor})}{2}, x \in [-2^{E-1}+1, 2^{E-1}] : \{x, E, P\} \in \mathbb{N} \quad (4.1)$$

O valor do erro associado ao arredondamento de um número real à representação do número de vírgula flutuante correspondente, segundo o standard IEEE 754, tem um comportamento proporcionalmente direto à grandeza do primeiro. Isto é, quanto maior for o número real a ser representado como *float* ou *double*, maior será o erro associado à operação de arredondamento da sua conversão, de tal modo que:

$$Erro(x) < Erro(y), \quad se \quad x < y \quad (4.2)$$

Por isso, tendo em conta a profundidade de cor de oito bits dos formatos de pixeis do modelo de cor YUV considerados neste trabalho, o maior valor a ser convertido para um dos tipos de números de vírgula flutuante é igual a $2^8 - 1 = 255$. Então, os maiores valores de erros associados à operação de arredondamento da conversão dos valores de intensidade de cor dos pixeis para os tipos de dados *float* e *double* são, respetivamente:

$$Erro_{float}(255) = \frac{ULP_{float}(255)}{2} = 2^{-16}$$
$$Erro_{double}(255) = \frac{ULP_{double}(255)}{2} = 2^{-45} \tag{4.3}$$

4.4.3 Análise Numérica da Operação de Reamostragem e Redimensionamento

Como referido nas secções 4.2 e em 4.3, apenas a operação de reamostragem e redimensionamento é realizada através de operações aritméticas, pois a operação de conversão do formato de pixeis de uma imagem envolve apenas operações lógicas e a reorganização da memória. Por essas razões, a análise numérica de toda a solução pode ser efetuada analisando numericamente a operação de reamostragem e redimensionamento.

Seguindo a formulação da operação de reamostragem de imagens como convolução, apresentada pela equação (3.10), o valor de intensidade de cor de um pixel de uma imagem reamostrada é obtido através do somatório do produto dos filtros de reconstrução e os valores de intensidade de cor dos a^2 pixeis mais próximos da imagem original, sendo a o valor de apoio de pixeis do filtro de reconstrução.

Assim, o erro de arredondamento da conversão de números reais aos tipos *float* e *double* associados aos resultados da operação de convolução de reamostragem é formulado matematicamente pela equação (4.4), sendo Δ o valor de incerteza associado ao resultado de uma função.

$$\Delta I_t(x,y) = \sum_{i=\lfloor u \rfloor - a + 1}^{\lfloor u \rfloor + a} \sum_{j=\lfloor v \rfloor - a + 1}^{\lfloor v \rfloor + a} Erro(I_s(\lfloor u \rfloor, \lfloor v \rfloor)) \times \Delta k(u-i) \times \Delta k(v-j) \tag{4.4}$$

Como $I_s(x,y)$ é a função de mapeamento dos pixeis da imagem original, o valor de incerteza associado ao resultado dessa função é o valor apresentado pelo racional da equação (4.3).

4.4.3.1 Análise Numérica dos Filtros de Reconstrução

O parâmetro dos filtros de reconstrução é a distância entre a posição do pixel da imagem reamostrada no sistema de coordenadas da imagem original e as posições dos a^2 pixeis mais próximos na imagem original. Assim, o valor do parâmetro de $k(x)$ está compreendido entre os valores do intervalo $[-\frac{a}{2}, \frac{a}{2}]$. Como os filtros de reconstrução lidam com as distâncias entre os pixeis, o valor absoluto do seu parâmetro de entrada poderá tomar os valores do intervalo: $[0, \frac{a}{2}]$.

Com o filtro de reconstrução *nearest neighbor*, sendo $a = 2$ o valor de apoio de pixeis, o parâmetro de entrada pode tomar valores do intervalo $[0,1]$. O valor de incerteza dos resultados do filtro, devido à propagação dos erros de arredondamento associados à utilização dos tipos *float* e *double*, é formulado como:

$$\Delta k_{nn}(x) = \begin{cases} Erro(1) & x < 0.5 \\ Erro(0) & senão \end{cases} \tag{4.5}$$

Assim, o valor de incerteza dos resultados deste filtro, assumindo o pior cenário possível, é:

$$\Delta k_{nn}(x) = max(Erro(1), Erro(0)) = Erro(1) \tag{4.6}$$

Quanto ao filtro de reconstrução linear, o valor de $a = 2$ implica que o parâmetro de entrada toma valores do intervalo $[0,1]$. O valor de incerteza deste filtro de reconstrução é dado como:

$$\Delta k_{linear}(x) = \begin{cases} \Delta(1-x) & x < 1 \\ Erro(0) & senão \end{cases} \tag{4.7}$$

$$\Delta k_{linear}(x) = max(\Delta(1-x), Erro(0)) = \Delta(1-x) = \sqrt{Erro(1)^2 + Erro(x)^2} \tag{4.8}$$

Por último, o filtro de reconstrução por *Spline* de Mitchell e Netravali é caraterizado pelo valor de apoio de pixeis igual a $a = 4$. Este valor indica que o parâmetro de entrada do filtro de reconstrução toma valores do intervalo $[0,2]$. A seguinte equação formula a incerteza do valor resultante do filtro:

$$\Delta k_{spline}(x) = \begin{cases} \Delta(1.4x^3 - 2.4x^2 + 1) & x < 1 \\ \Delta(-0.6x^3 + 3x^2 - 4.8x + 2.4) & 1 \leq x < 2 \\ Erro(0) & senão \end{cases} \tag{4.9}$$

Os valores de incerteza de cada um dos ramos do filtro de reconstrução considerado na equação anterior, são demonstrados matematicamente pelas equações apresentadas de seguida através do racional teórico de análise de propagação de erros descrito em [44].

$$\begin{gathered} \Delta(1.4x^3 - 2.4x^2 + 1) = \\ (\Delta(1.4x^3 - 2.4x^2)^2 + Erro(1)^2)^{\frac{1}{2}} = \\ (\Delta(1.4x^3)^2 + \Delta(-2.4x^2)^2 + Erro(1)^2)^{\frac{1}{2}} = \\ ((1.4 \times \Delta(x^3) + x^3 \times Erro(1.4))^2 + (-2.4 \times \Delta(x^2) + x^2 \times Erro(-2.4))^2 + Erro(1)^2)^{\frac{1}{2}} = \\ ((\frac{4.2 \times Erro(x)}{x} + x^3 \times Erro(1.4))^2 + (\frac{-4.8 \times Erro(x)}{x} + x^2 \times Erro(-2.4))^2 + Erro(1)^2)^{\frac{1}{2}} \end{gathered} \tag{4.10}$$

Similarmente, aplicando a análise no ramo complementar do mesmo filtro de reconstrução:

$$\begin{aligned}\Delta(-0.6x^3+3x^2-4.8x+2.4) =&((\frac{-1.8\times Erro(x)}{x}+x^3\times Erro(-0.6))^2+\\ &(\frac{6\times Erro(x)}{x}+x^2\times Erro(3))^2+\\ &(-4.8\times Erro(x)+x\times Erro(-4.8))^2+\\ &Erro(2.4)^2)^{\frac{1}{2}}\end{aligned} \tag{4.11}$$

Experimentalmente verificou-se a desigualdade $\Delta(1.4x^3-2.4x^2+1)<\Delta(-0.6x^3+3x^2-4.8x+2.4)$ para valores de x pertencentes ao intervalo $[0,2]$, por isso pode-se concluir que:

$$\begin{aligned}\Delta k_{spline}(x) =&max(\Delta(1.4x^3-2.4x^2+1),\\ &\Delta(-0.6x^3+3x^2-4.8x+2.4),\\ &Erro(0)) =\\ &\Delta(-0.6x^3+3x^2-4.8x+2.4)\end{aligned} \tag{4.12}$$

Em suma, a incerteza do resultado de cada um dos filtros de reconstrução, devido à propagação de erros causados pelo arredondamento de números reais aos tipos de dados de números de vírgula flutuante, *float* e *double*, são apresentadas na tabela 4.3:

Tabela 4.3: Incerteza dos resultados dos filtros de reconstrução.

	k_{nn}	k_{linear}	k_{spline}
Float	1.19×10^{-7}	1.68×10^{-7}	6.90×10^{-4}
Double	2.22×10^{-16}	3.14×10^{-16}	2.98×10^{-8}

4.4.3.2 Análise Numérica da Convolução

Como o cálculo do valor de intensidade de cor de um pixel pela operação de reamostragem é obtido pela a aplicação do filtro de reconstrução nas duas dimensões da imagem, na horizontal e vertical, a incerteza associada ao produto dos resultados dos filtros de reconstrução nas duas dimensões é:

$$\Delta(k(u-i)\times k(v-j)) = k(u-i)\times\Delta(k(v-j))+k(v-j)\times\Delta(k(u-i)) \tag{4.13}$$

O valor resultante dos filtros de reconstrução é um valor normalizado pertencente ao intervalo $[0,1]$. Tendo em conta a análise de incerteza da secção 4.4.3.1, os valores de incerteza associados aos resultados do produto dos filtros de reconstrução em ambas dimensões são apresentados na tabela 4.4:

Tabela 4.4: Incerteza do produto dos filtros de reconstrução.

	$k_{nn} \times k_{nn}$	$k_{linear} \times k_{linear}$	$k_{spline} \times k_{spline}$
Float	2.38×10^{-7}	3.37×10^{-7}	1.38×10^{-3}
Double	4.44×10^{-16}	6.28×10^{-16}	5.96×10^{-8}

Como explicado na secção 4.4.3, a incerteza associada à função $I_s(\lfloor u \rfloor, \lfloor v \rfloor)$ é igual ao valor de $Erro(255)$. Assim, os valores de incerteza associados ao valores de intensidades de cor ponderados dos pixeis da imagem original pelo resultado dos filtros de reconstrução são dados por:

$$\Delta(I_s(\lfloor u \rfloor, \lfloor v \rfloor) \times \Delta(k(u-i) \times k(v-j))) = \tag{4.14}$$
$$I_s(\lfloor u \rfloor, \lfloor v \rfloor) \times \Delta(k(u-i) \times k(v-j)) + \Delta(I_s(\lfloor u \rfloor, \lfloor v \rfloor)) \times k(u-i) \times k(v-j)$$

Os resultados dos valores de incerteza da equação (4.14) são apresentados na tabela 4.5:

Tabela 4.5: Incerteza do valor de intensidade de cor de um pixel da convolução.

	$I_s \times k_{nn} \times k_{nn}$	$I_s \times k_{linear} \times k_{linear}$	$I_s \times k_{spline} \times k_{spline}$
Float	7.60×10^{-5}	1.01×10^{-4}	3.52×10^{-1}
Double	1.41×10^{-13}	1.88×10^{-13}	1.51×10^{-5}

Tendo em conta que a operação de convolução sobre os a^2 pixeis da região de apoio de pixeis caraterística de cada filtro de reconstrução, o valor de incerteza associado ao valor de intensidade de cor obtido pela operação de convolução de um pixel da imagem reamostrada é apresentado na tabela 4.6:

Tabela 4.6: Incerteza do valor de intensidade de cor de um pixel da convolução.

	$\sum^a \sum^a I_s \times k_{nn} \times k_{nn}$	$\sum^a \sum^a I_s \times k_{linear} \times k_{linear}$	$\sum^a \sum^a I_s \times k_{spline} \times k_{spline}$
Float	1.52×10^{-4}	2.02×10^{-4}	1.408
Double	2.83×10^{-13}	3.77×10^{-13}	6.07×10^{-5}

4.4.3.3 Conclusão da Análise Numérica

A análise numérica do processo de reamostragem e redimensionamento permitiu traçar uma paralelização entre os dois diferentes tipos de dados utilizados para a representação de números de vírgula flutuante. Os tipos *float* e *double* têm um valor de incerteza associado ao arredondamento necessário para a representação de um número real como um número de vírgula flutuante, que se propagam pelas operações que utilizam estes tipos de dados.

A tabela 4.6 expõe os valores de incerteza associados aos resultados do processo de reamostragem e redimensionamento de imagem relativamente ao tipo de dados e filtro de reconstrução utilizados.

Pelos resultados apresentados na tabela 4.6 é possível concluir que o erro de precisão do tipo *double* não tem impacto no valor de intensidade de cor do pixel da imagem reamostrada obtido pela operação de convolução, pois como os valores de intensidade de cor dos pixeis pertencem ao conjunto de números naturais, incertezas inferiores a uma unidade não são suficientes para modificar os resultados.

Contudo, a utilização do tipo *float*, em especifico no processo de reamostragem e redimensionamento com o filtro de reconstrução por *Spline*, apresenta um valor de incerteza superior a uma unidade. Essa ocorrência influencia os resultados do processo de reamostragem, o que provoca que o valor de intensidade de cor do pixel da imagem reamostrada obtido seja diferente do valor esperado. No entanto, devido às elevadas resoluções de imagem utilizadas atualmente, a visão humana não tem perceção suficiente para distinguir as diferenças das caraterísticas e detalhes da imagem com apenas uma unidade de diferença [45].

Capítulo 5

Resultados

O objetivo da solução implementada neste trabalho consiste na redução dos tempos de execução no processo de reamostragem e redimensionamento em vídeo sem compressão face à solução existente baseada em FFmpeg. O capítulo descreve os testes realizados, a metodologia da análise, e os resultados obtidos comparativamente entre a solução implementada e o FFmpeg.

5.1 Metodologia

De modo a poder avaliar a solução implementada neste trabalho relativamente aos objetivos do mesmo, procedeu-se à medição do tempo de execução do processo de reamostragem e redimensionamento de um vídeo sem compressão atendendo à divisão de operações apresentadas no capítulo 4.

Tendo em conta que a operação de reamostragem e redimensionamento de uma imagem é constituída por operações intermédias de conversão de formatos de pixeis, o desempenho observado nas operações de conversão influencia o desempenho da operação de reamostragem. A análise dos resultados da solução implementada é, então, dividida na análise das duas operações que constituem o processo de reamostragem e redimensionamento de um vídeo.

As operações de conversão do tipo de formatos de pixeis e de reamostragem e redimensionamento de frames são analisadas independentemente em termos de ganho de desempenho em comparação entre a solução desenvolvida e a ferramenta FFmpeg.

Para simular as condições impostas à ferramenta de pós-produção de conteúdos de multimédia onde a solução implementada será integrada, foi utilizado um vídeo sem compressão segundo o modelo de cor YUV com diferentes tipos de formatos de pixeis.

O vídeo utilizado é um filme de livre uso intitulado *Big Buck Bunny*, amplamente utilizado para testar operações de processamento de vídeo e imagem, e por isso é considerado o caso de estudo standard deste tipo de operações.

Figura 5.1: Frame do filme *Big Buck Bunny* em *FHD*.

Este filme, totalmente criado digitalmente, é fornecido segundo o modelo de cor RGB em vários tipos de compressão. O filme sem compressão é caraterizado pela profundidade de cor de 8 bits e um tipo de subamostragem de 4:4:4. O filme é constituído por 14 315 frames, considerando uma frame *rate* de apresentação do vídeo de 24 FPS, a sua duração é de 9 minutos e 57 segundos. Para uma resolução *full high definition*, ou *FHD*, o filme ocupa um tamanho de 28 GB (29 487 MB) em memória.

O vídeo *Big Buck Bunny*, com as caraterísticas já apresentadas, foi convertido para o modelo de cor YUV com representações variadas de formatos de pixeis e diferentes tipos de subamostragem de crominâncias para simular as mesmas caraterísticas dos conteúdos multimédia processados pela ferramenta onde a solução desenvolvida será integrada.

A análise de desempenho do processo de reamostragem e redimensionamento de vídeo irá apenas considerar o tempo de execução da aplicação das operações de conversão de formato de pixeis e de reamostragem. As restantes operações como a leitura e escrita, respetivamente, da frame original do vídeo e reamostrada, são omitidas da análise realizada visto que o tempo de processamento da execução destas operações é irrelevante por se tratar de *overhead* causado pela implementação dos cenários de teste e não ser o foco de estudo deste trabalho.

5.1.1 Métrica de Avaliação

O desempenho analisado da solução desenvolvida neste trabalho é realizado através da métrica de *speed up*. O *speed up* é uma métrica baseada na noção definida pela lei de Amdahl e é utilizada, geralmente, para analisar o tempo de execução da solução de um problema em relação a outra solução.

Considerando o tempo de execução de duas soluções diferentes para o mesmo problema L_a e L_b, o valor de *speed up* da solução a, caraterizado pela variável S_a, é obtido através da formulação:

$$S_a = \frac{L_b}{L_a} \tag{5.1}$$

No contexto deste problema, como se pretende analisar o desempenho da solução desenvolvida neste trabalho em relação à ferramenta FFmpeg, o valor do tempo de execução de referência

pertence à ferramenta FFmpeg e o valor do tempo de execução da solução implementada é o valor essencial da métrica *speed up* que se pretende calcular.

Sendo L_{FFmpeg} o valor do tempo de execução da ferramenta FFmpeg e $L_{solucao}$ o valor do tempo de execução da solução desenvolvida neste trabalho, o desempenho ganho pela última é dado através da métrica *speed up* da seguinte equação:

$$S_{solucao} = \frac{L_{FFmpeg}}{L_{solucao}} \tag{5.2}$$

5.1.2 Ambiente de Teste

A análise do desempenho da solução desenvolvida neste trabalho foi efetuada utilizando diferentes ambientes de execução de modo a poder inferir as suas limitações e aptidões em termos de processamento.

A operação de conversão de formato de pixeis de imagens, executado exclusivamente pelas capacidades computacionais do CPU, foi analisada em duas máquinas com diferentes caraterísticas como o número de núcleos de processamento, o valor de frequência de relógio do CPU e a capacidade de armazenamento do sistema de memória cache. A tabela 5.1 apresenta as especificação das caraterísticas referidas das diferentes máquinas de teste:

Tabela 5.1: Comparação de especificações entre os processadores utilizados.

	Máquina 1 (M1)	Máquina 2 (M2)
Procesador	Intel Core i7-4770K	Intel Xeon E5-2640 v4
Cores	4 (8 Threads)	10 (20 Threads)
Frequência (GHz)	Base: 3,5 - Turbo: 3,9	Base: 2,4 - Turbo: 3,4
Cache	8 MB SmartCache	25 MB SmartCache

A máquina M2 é constituída por dois processadores do modelo apresentado. Por essa razão, o número de núcleos de processamento da máquina é, na prática, o dobro do valor apresentado na tabela.

A operação de reamostragem e redimensionamento de frames de um vídeo foi analisada com diferentes unidades de processamento gráfico na mesma máquina, M1. Embora similares em termos de arquitetura base, os GPUs testados diferem em termos de capacidade computacional devido às diferentes caraterísticas especificadas na tabela 5.2:

Tabela 5.2: Comparação de especificações entre as unidades de processamento gráficas utilizadas.

	GPU 1	GPU 2
Modelo	NVidia Quadro P600	NVidia Quadro P2000
Cores	384	1024
Performance Máxima FP32 (TFLOPS)	1,195	3,0
Memória Integrada	2 GB GDDR5	5 GB GDDR5
Largura de Banda de Memória	64 GB/s	140 GB/s

5.2 Resultados da Solução Desenvolvida Relativamente ao FFmpeg

Como referido no capítulo 4 de descrição da solução implementada, o processo de reamostragem e redimensionamento de vídeo está dividido em duas operações, a operação de conversão de formatos de pixeis das frames e a operação da sua reamostragem.

De modo a fazer uma análise aprofundada de todo o processo, cada uma das operações será analisada individualmente em relação à aplicação dos mesmos processos utilizando a ferramenta FFmpeg.

5.2.1 Detalhes

Antes de iniciar a análise da solução desenvolvida relativamente ao FFmpeg é necessário compreender o comportamento e detalhes do último para a aplicação das operações que constituem o processo de reamostragem e redimensionamento de vídeo.

Para vídeo sem compressão, de formato *raw*, o FFmpeg utiliza um conjunto de métodos do seu módulo *libswscale* específicos para o processamento de vídeo deste formato. Contudo, os métodos de processamento de vídeo sem compressão têm um funcionamento sequencial utilizando apenas um núcleo de processamento do CPU.

Embora os métodos de processamento do tipo de vídeo considerado sejam sequenciais com a ferramenta FFmpeg, a última tira partido de um mecanismo de aceleração do CPU para reduzir o tempo de execução dos métodos de processamento, designado de vetorização de operações.

Este tipo de mecanismo implementa implicitamente a paralelização de métodos no CPU através da especificação de instruções especiais, denominadas de *intrinsics*, que indicam ao compilador a execução concorrente de instruções sobre um conjunto de dados. A execução de código vetorizado tem um melhor desempenho em relação ao processamento sequencial do mesmo código [46].

5.2.2 Análise da Operação de Conversão de Formato de Pixeis

A operação de conversão de formato de pixeis de imagens, executada exclusivamente pelo CPU, foi analisada nas diferentes máquinas M1 e M2. Os testes efetuados envolveram a aplicação da operação em frames do vídeo *Big Buck Bunny*, segundo o modelo de cor YUV, na resolução *8K*, ou 7680×4320.

A resolução *8K* de vídeo foi utilizada na análise da operação de conversão do formato de pixeis de frames pois apresenta uma maior discrepância entre os resultados quer das diferentes máquinas onde foram executados os testes, quer da solução utilizada.

Nesta secção são expostas figuras de gráficos que apresentam os resultados da operação de conversão de pixeis dependendo do formato de pixeis de entrada, o formato de pixeis de saída, a máquina onde foi executado o teste e que solução foi utilizada, a ferramenta FFmpeg ou a solução desenvolvida neste trabalho.

A figura 5.2 expõe os resultados do tempo de execução da operação considerada em milissegundos, sendo o principal foco desta representação a comparação dos tempos de execução entre as diferentes combinações de tipos de entrada e saída de formatos de pixeis em diferentes máquinas utilizando apenas um *core*.

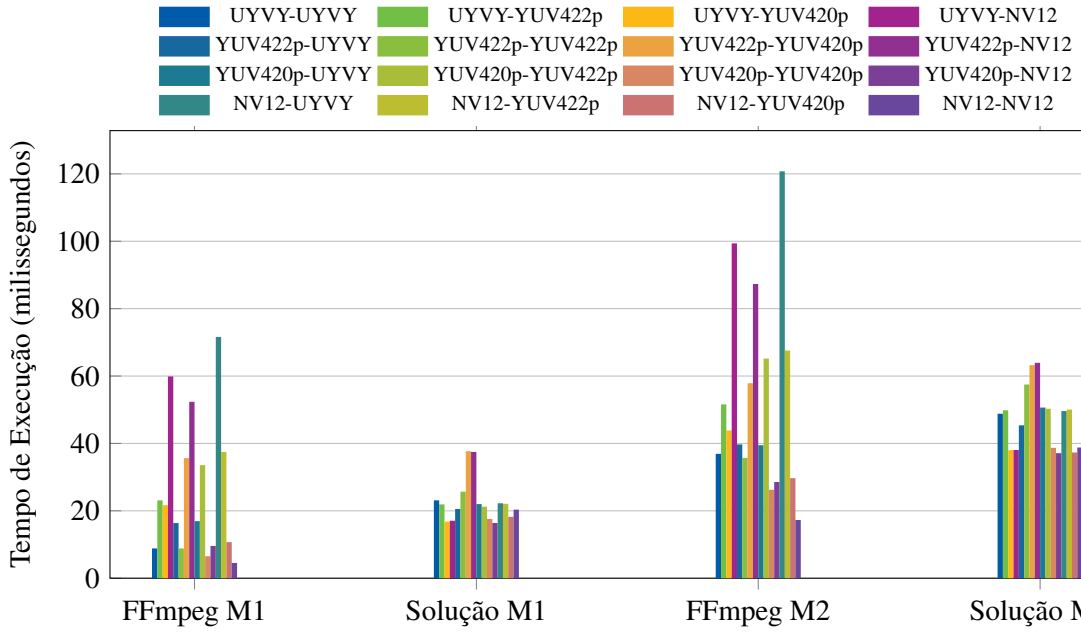

Figura 5.2: Tempo de execução da operação de conversão do tipo de formatos de pixeis de imagens *8K* em diferentes máquinas utilizando 1 *core*.

Como é possível observar, de forma geral, os tempos de execução da operação de conversão de formato de pixeis na máquina M2 são piores do que os observados na máquina M1, tanto quanto a ferramenta FFmpeg como a solução implementada neste trabalho, utilizando apenas 1 *core* para o processamento. Esta ocorrência deve-se ao menor valor de frequência de relógio do modelo do CPU da máquina M2 em comparação com a máquina M1.

Inicialmente, a implementação da solução deste trabalho definia a utilização máxima do CPU das máquinas de teste durante a execução da operação, através da utilização completa de todos os núcleos de processamento. Contudo, nessas condições, o tempo de execução da solução implementada para a operação de conversão de formato de pixeis na máquina M2 eram superiores aos tempos de execução utilizando um menor número de *cores*.

Após um estudo sobre o impacto do número de núcleos de processamento na aplicação da operação em análise com a solução proposta neste trabalho na máquina M2, foi possível perceber que a redução do desempenho da operação se devia a um aumento do número de *page faults* do sistema de memória cache. Quanto maior o número de núcleos de processamento, maior o número de *threads* que realizam acessos a memória em simultâneo. Os acessos a posições díspares da memória implicam uma constante atualização dos valores armazenados no sistema de memória cache devido à sua reduzida capacidade de armazenamento.

De modo a investigar a partir de que número de núcleos de processamento, utilizados na aplicação da operação de conversão de formato de pixeis, existia um impacto negativo de desempenho, foram executados testes que utilizavam diferentes números de *cores* da máquina M2. Os resultados deste estudo encontram-se no gráfico da figura 5.3:

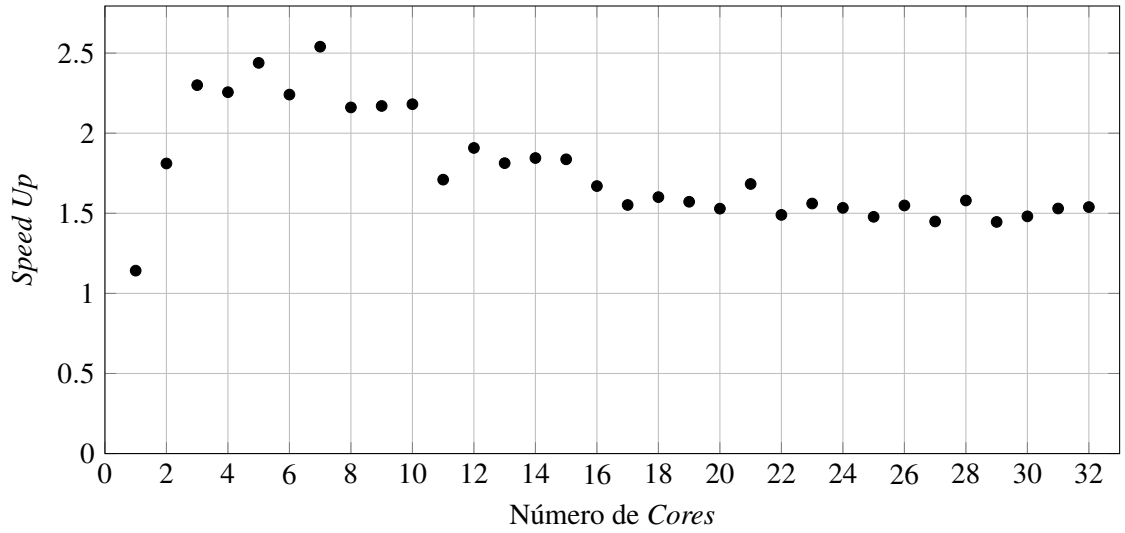

Figura 5.3: Variação do valor de *speed up* da solução proposta em relação à ferramenta FFmpeg em função do número de núcleos de processamento utilizados na máquina M2.

É possível concluir que a operação de conversão do tipo de formato de pixeis de imagem da solução implementada neste trabalho não apresenta um tempo de execução melhorado quanto maior o número de núcleos de processamento utilizados. Observando a figura 5.3 nota-se um pico do valor de *speed up* em relação à ferramenta FFmpeg com a utilização de apenas 7 *cores* no processamento de uma imagem. Um acréscimo do número de *cores* utilizados superior a 7 piora o valor de *speed up* da solução associado à máquina M2, pois a operação considerada fica condicionada pelo desempenho da memória relativamente à latência de respostas a acessos realizados.

Após este estudo, a operação de conversão de formato de pixeis da solução desenvolvida neste trabalho tem uma condição que limita o número de núcleos de processamento utilizados. Se a máquina possuir um número de *cores* superior a 7, o número de núcleos de processamento utilizados ficará limitado a esse número; caso contrário, a máquina utiliza completamente todos os *cores* do seu CPU.

As tabelas B.1 e B.2 em apêndice apresentam os resultados em milissegundos de cada uma das operações de conversão de formatos de pixeis de uma imagem em função da solução utilizada e a máquina utilizada para a sua execução com 7 *cores*. Estas tabelas permitem explorar os resultados apresentadas no gráfico da figura 5.4.

A figura 5.4 expõe os resultados do tempo de execução da operação de conversão de formato de pixeis para formatos de entrada e saída diferentes obtidos pela execução da ferramenta FFmpeg e a solução desenvolvida neste trabalho em diferentes máquinas de teste.

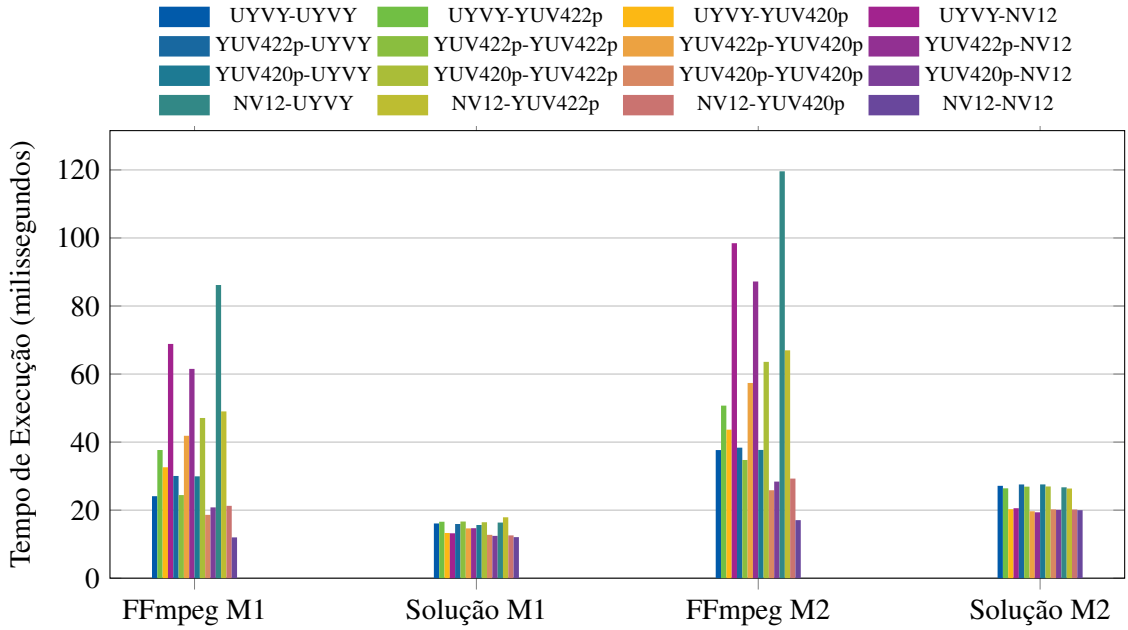

Figura 5.4: Tempo de execução da operação de conversão do tipo de formatos de pixeis de imagens *8K* em diferentes máquinas utilizando 7 *cores*.

Com esta análise verificou-se um melhor desempenho em termos de tempo de processamento na execução da solução proposta neste trabalho em diferentes máquinas em comparação com a ferramenta FFmpeg. Também, a máquina M1 apresenta um tempo de execução inferior relativamente a máquina M2 na aplicação da operação de conversão de formato de pixeis de frames de um vídeo. Por essa razão, os restantes testes realizados a partir deste ponto consideram apenas a máquina M1.

5.2.3 Análise da Operação de Reamostragem e Redimensionamento

A operação de reamostragem e redimensionamento de frames de um vídeo foi analisada na máquina M1, devido ao seu melhor desempenho da operação de conversão de formato de pixeis, com diferentes unidades de processamento gráfico, o GPU1 e o GPU2. Os testes efetuados envolveram a aplicação da operação de reamostragem em frames do vídeo *Big Buck Bunny* segundo o modelo de cor YUV na resolução *FHD*.

Devido aos diferentes filtros de reconstrução existentes para a aplicação do processo de reamostragem de imagens e à complexidade das suas operações aritméticas, os testes foram divididos em grupos diferentes. Os tempos de execução da operação de reamostragem e redimensionamento diferem em função do filtro de reconstrução e as capacidades computacionais do GPU utilizados. Por essa razão, os resultados dos testes efetuados foram agrupados pelos diferentes filtros de reconstrução considerando as múltiplas resoluções da frame reamostrada.

De modo a ter uma visão geral do comportamento da operação de reamostragem e redimensionamento de um vídeo, foram considerados apenas a reamostragem de frames da resolução *FHD*

para *576p* e *720p*, respetivamente 1024×576 e 1280×720, para simular o *downscaling* de vídeo, e a reamostragem de frames da resolução *FHD* para *1440p* e *4K*, respetivamente 2560×1440 e 3840×2160, para simular o *upscaling* de vídeo.

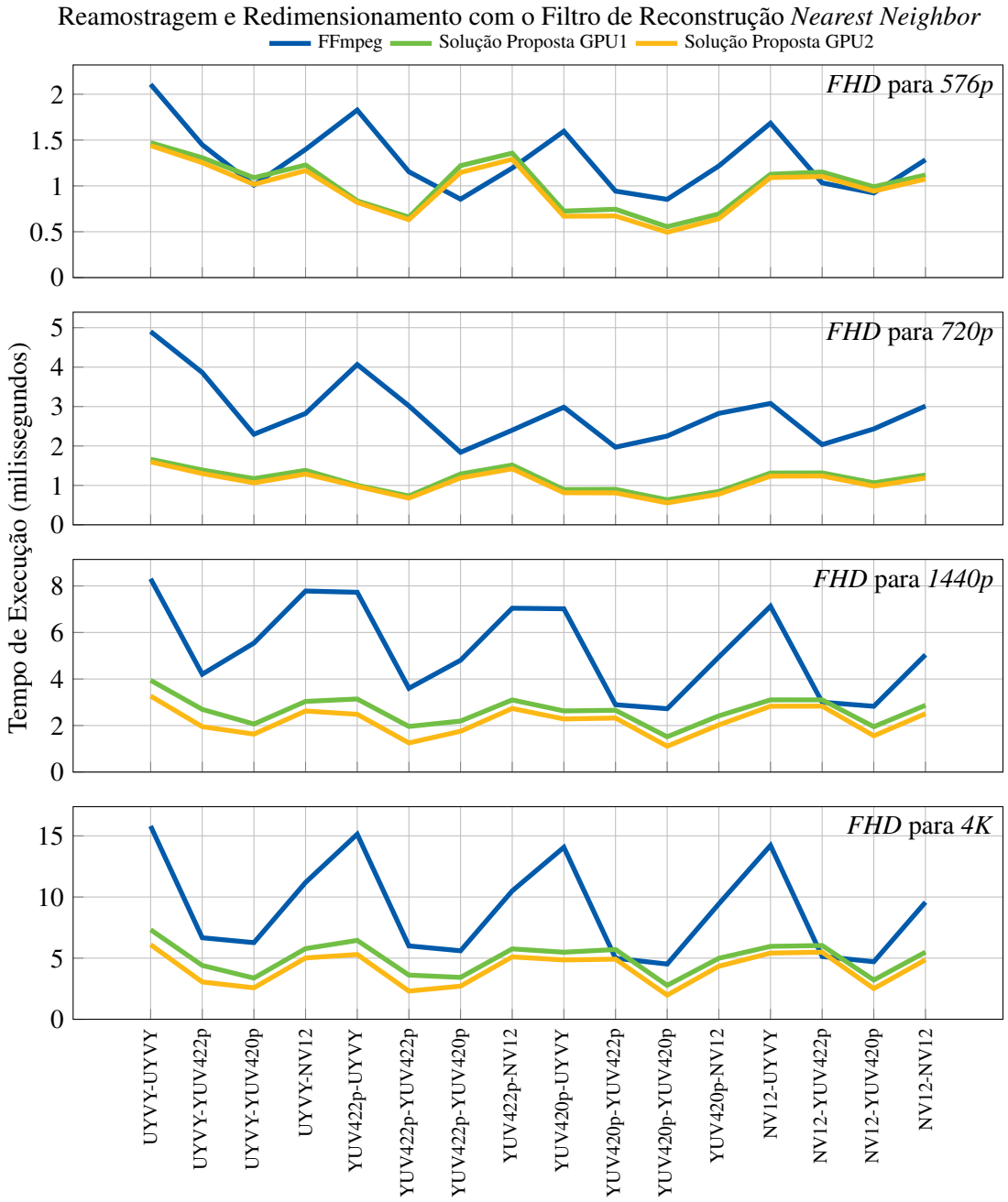

Figura 5.5: Tempos de execução em milissegundos da operação de reamostragem com o filtro de reconstrução *Nearest Neighbor* com diferentes condições de processamento.

A figura 5.5 expõe os tempos de execução em milissegundos da operação de reamostragem e redimensionamento de frames utilizando o filtro de reconstrução NN, ou *Nearest Neighbor*. Este

filtro envolve um reduzido número de operações aritméticas e, por essa razão, a operação neste tipo de condições não apresenta um considerável ganho de desempenho da solução proposta em relação à ferramenta FFmpeg. O ganho de performance da operação de reamostragem com o filtro de reconstrução NN da solução proposta, relativamente à ferramenta FFmpeg, deve-se ao melhor desempenho obtido pelas operações de conversão de formato de pixeis intermédias e a reduzida latência dos acessos a memória do GPU.

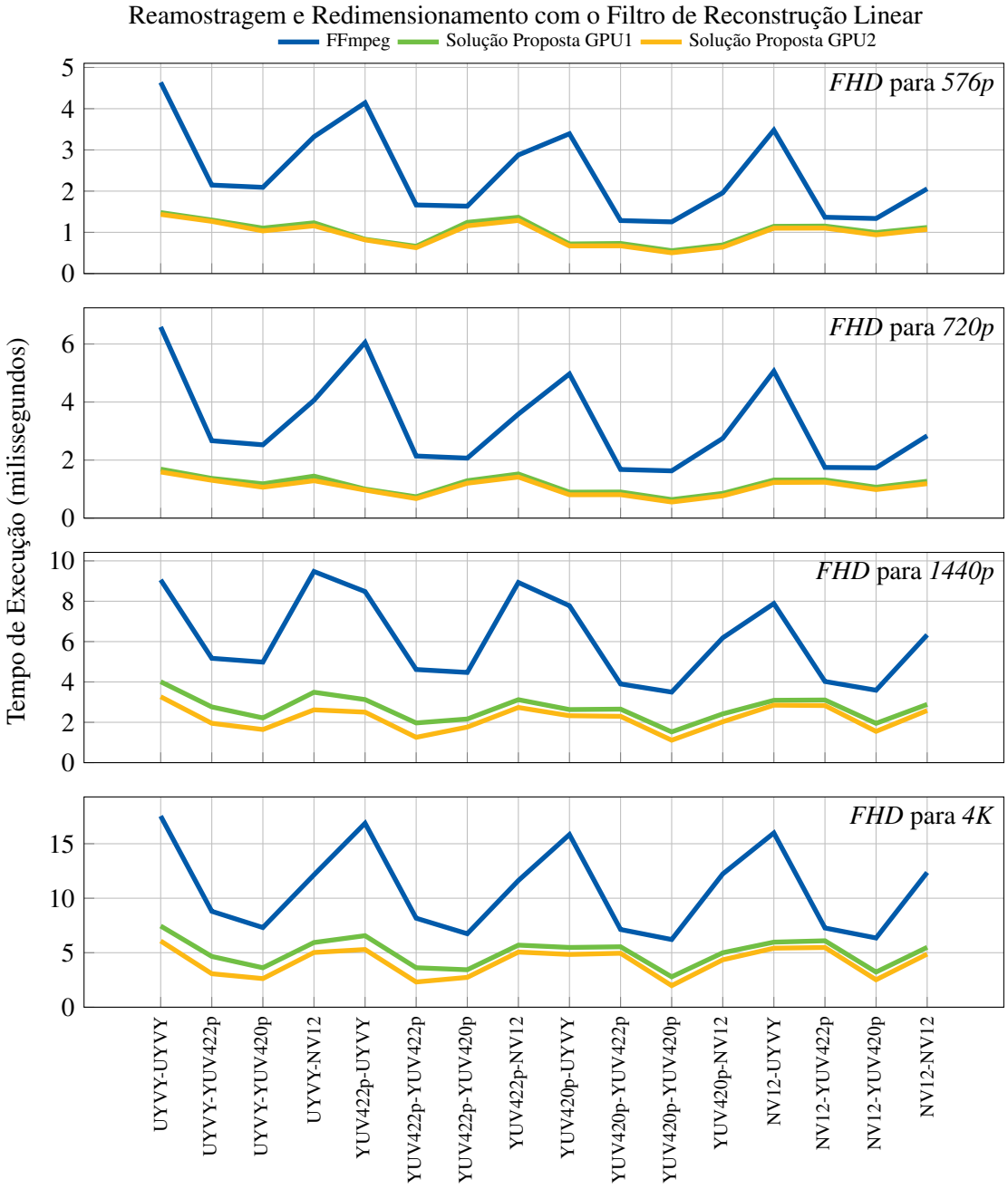

Figura 5.6: Tempos de execução em milissegundos da operação de reamostragem com o filtro de reconstrução linear com diferentes condições de processamento.

A figura 5.6 exibe os tempos de execução da aplicação do filtro de reconstrução linear na operação de reamostragem e redimensionamento. O filtro linear é constituído por um maior número de operações aritméticas em comparação com o filtro NN. Por essa razão, é possível tirar um melhor partido das capacidades computacionais do GPU, o que se reflete na maior diferença dos tempos de processamento entre a ferramenta FFmpeg e a solução desenvolvida.

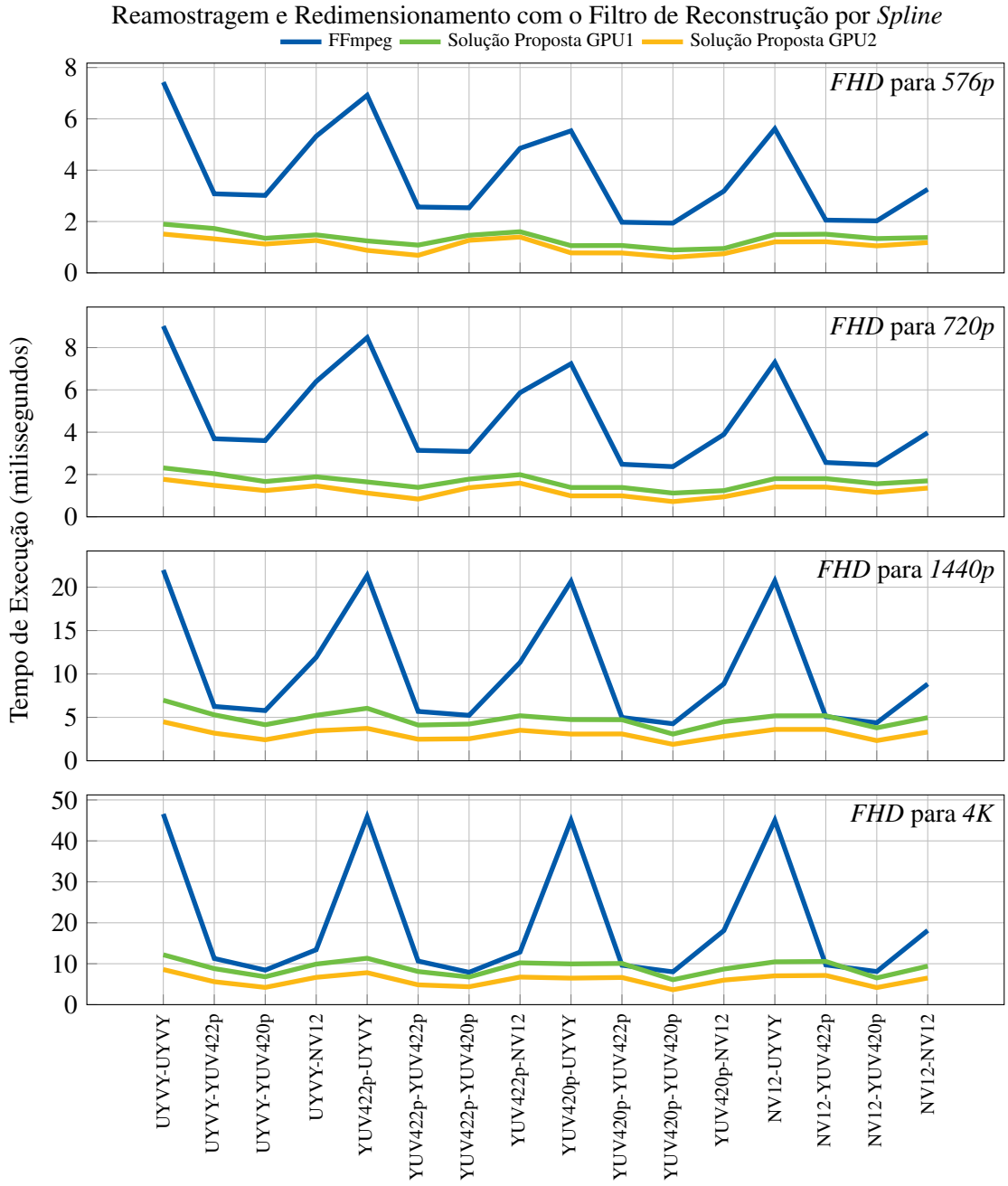

Figura 5.7: Tempos de execução em milissegundos da operação de reamostragem com o filtro de reconstrução por *spline* com diferentes condições de processamento.

A figura 5.7 expõe os tempos de execução da aplicação do filtro de reconstrução por *spline* na operação de reamostragem e redimensionamento de imagem. O filtro de reconstrução por *spline* tem um imenso impacto no tempo de execução da interpolação desta operação, pois é realizado através de múltiplos acessos a memória e operações aritméticas, relativamente aos filtros *Nearest Neighbor* e linear. Por essa razão, os resultados do filtro de reconstrução por *spline* pertencem a uma magnitude de valores superiores relativamente aos restantes filtros.

De modo a finalizar a análise dos resultados da operação de reamostragem e redimensionamento de frames de um vídeo, apresentados nas figuras 5.5, 5.6 e 5.7, são retiradas as conclusões referidas de seguida. Os tempos de execução da operação de reamostragem resultado da aplicação da solução desenvolvida neste trabalho pelas unidades de processamento gráfico apresentam as mesmas variações de valores. Em cada teste de reamostragem entre duas resoluções por cada filtro de reconstrução, o GPU1 e GPU2 seguem a mesma tendência de valores de tempo de execução diferindo entre si em alguns milissegundos. Esta diferença é mais percetível quanto maior o número de acessos a memória necessários a serem realizados. Devido à maior largura de banda reservada à transferência de dados de memória do GPU2, a latência de respostas a acessos de memória do GPU2 é menor que a latência apresentada pelo GPU1. Por essa razão, a diferença dos tempos de execução das duas unidades de processamento gráfico é maior quanto maior o valor de apoios de pixeis do filtro de reconstrução utilizado.

De uma forma generalizada o tempo de execução obtido com a ferramenta FFmpeg na aplicação do processo de reamostragem é superior aos tempos de execução obtidos por ambas unidades de processamento gráfico com a solução proposta neste trabalho. É possível observar pelas figuras 5.5, 5.6 e 5.7 que quanto maior o número de operações aritméticas a serem realizadas, isto é, quanto maior a complexidade do cálculo de resultados dos filtros de reconstrução, melhor é a vantagem de utilização dos GPUs na operação de reamostragem. Esta afirmação é corroborada pela crescente diferença entre os tempos de execução, quanto maior a complexidade do filtro de reconstrução utilizado, obtidos pela ferramenta FFmpeg e a solução implementada neste trabalho. Por último, considerando apenas um determinado teste da operação de reamostragem de uma frame entre duas resoluções e um filtro de reconstrução em específico, é possível distinguir os resultado do tempo de execução em dois grupos de tipos de formatos de pixeis. Para formatos de pixeis da frame de saída não planares, como os tipos entrelaçados e semi-planares, os tempos de execução obtidos pela ferramenta FFmpeg são consideravelmente mais elevados do que os obtidos com os tipos de formatos planares. Esta situação deve-se às caraterísticas de implementação do FFmpeg.

A ferramenta FFmpeg não implementa um tipo de formato de pixel intermédio para realizar a operação de reamostragem de uma frame, como o que acontece na solução proposta deste trabalho. Por essa razão, a operação de reamostragem aplicada pelo FFmpeg é efetuada sobre o tipo de formato de pixel da frame reamostrada. O processamento de vídeo sobre os tipos de formatos de pixeis entrelaçados e semi-planares implica um maior número de operações lógicas de modo a separar temporariamente as componentes de cor da imagem e processa-las individualmente. Por consequência, o processamento realizado sobre este tipo de formatos de pixeis acarreta *overhead* adicional de processamento, como pode ser observado pelas figuras 5.5, 5.6 e 5.7.

5.2.4 Conclusões

Com a análise de resultados concluída, é possível definir os valores médios de ganho de performance da operação de conversão de formato de pixeis e da operação e reamostragem de vídeo da solução proposta neste trabalho relativamente à ferramenta FFmpeg. A operação de conversão do tipo de formatos de pixeis, segundo a implementação da solução deste trabalho e imagens de resolução *8K*, resulta numa diminuição em aproximadamente 48% do tempo de execução da mesma operação por parte da ferramenta FFmpeg.

A tabela 5.3 apresenta os valores de redução, em percentagem, dos tempos de execução obtidos pela solução desenvolvida para a operação de reamostragem de vídeo em relação à ferramenta FFmpeg de um vídeo na resolução *FHD*. A tabela seguinte distingue a peculiaridade, referida na secção 5.2.3, de um maior ganho de desempenho na operação de reamostragem e redimensionamento para formatos de pixeis de saída do tipo não planar devido ao maior tempo de execução despendido pela ferramenta FFmpeg.

Tabela 5.3: Percentagem de redução do tempo de execução em milissegundos da aplicação da solução desenvolvida neste trabalho nos dois GPUs de teste relativamente aos resultados da ferramenta FFmpeg para um vídeo na resolução *FHD*.

	Planar				Não Planar			
NN	1024x576	1280x720	2560x1440	3840x2160	1024x576	1280x720	2560x1440	3840x2160
FFmpeg	1,027	2,463	3,701	5,486	1,539	3,262	6,874	12,491
Solução GPU1	-6,08%	-56,86%	-38,75%	-25,86%	-30,48%	-62,13%	-55,93%	-52,73%
Solução GPU2	-11,66%	-60,34%	-51,33%	-41,81%	-33,44%	-64,39%	-62,26%	-58,98%
Linear								
FFmpeg	1,598	2,023	4,282	7,247	3,233	4,488	8,016	14,325
Solução GPU1	-39,44%	-47,54%	-46,45%	-43,10%	-66,81%	-72,20%	-61,37%	-58,48%
Solução GPU2	-42,83%	-51,64%	-57,95%	-55,71%	-68,37%	-74,24%	-67,39%	-64,28%
Spline								
FFmpeg	2,398	2,925	5,206	9,216	5,263	6,520	15,707	30,616
Solução GPU1	-45,69%	-45,51%	-17,06%	-13,61%	-73,63%	-73,22%	-65,93%	-66,44%
Solução GPU2	-58,08%	-60,66%	-48,36%	-44,88%	-78,72%	-79,57%	-77,74%	-77,20%

A operação de reamostragem e redimensionamento de vídeo *FHD*, segundo a implementação da solução proposta neste trabalho, diminui, em média, o tempo de execução relativamente a ferramenta FFmpeg em aproximadamente 42% e 64% a operação de reamostragem no processamento de formatos de pixeis de saída do tipo planar e não planar, respetivamente. Este ganho de desempenho satisfaz um dos objetivos do trabalho desta dissertação que consiste na redução do tempo de execução despendido no processo de reamostragem e redimensionamento de vídeo em relação ao processo atual utilizado pela ferramenta FFmpeg.

Outro objetivo desta dissertação é o processamento em tempo real de vídeo de elevadas resoluções em *stream*. Considerando que os standards definem uma frame *rate* de captura de imagens de um vídeo entre os 24 e 60 frames por segundo [28], uma ferramenta é capaz de atingir o processamento real de vídeos quando o número de frames processadas por segundo ultrapassa o valor

da frame *rate* de captura.

O número de frames processadas por segundo de um vídeo depende da sua resolução e do filtro de reconstrução utilizado na convolução da operação de reamostragem. A tabela 5.4 apresenta os valores de processamento em FPS da ferramenta FFmpeg e a solução proposta em função dos filtros de reconstrução utilizados e a resolução do vídeo a ser reamostrado.

Tabela 5.4: Número de frames por segundo processadas pela ferramenta FFmpeg e a solução desenvolvida neste trabalho para vídeos de diferentes resoluções.

	Full High Definition			4K		
	Nearest Neighbor	Linear	*Spline*	*Nearest Neighbor*	Linear	*Spline*
FFmpeg	217	176	102	42	36	20
Solução	476	403	350	110	93	70

A versão atual da ferramenta FFmpeg atinge o processamento em tempo real de vídeo *FHD* em *stream*. Contudo, o mesmo não acontece para resoluções superiores como é o caso da resolução *4K*. Por outro lado, a solução desenvolvida neste trabalho é capaz de processar em tempo real vídeo em *stream* tanto para a resolução *Full High Definition* como *4K*.

Considerando o vídeo de teste *Big Buck Bunny*, com 14 315 frames e uma duração de 9 minutos e 57 segundos, o processo de reamostragem e redimensionamento do filme na resolução *4K* pelo filtro de reconstrução por *spline* tem uma duração de processamento de 11 minutos e 58 segundos com a ferramenta FFmpeg, o que não é considerado processamento em tempo real, e 3 minutos e 25 segundos pela solução proposta.

A utilização da solução desenvolvida neste trabalho ao invés da ferramenta FFmpeg traz outras vantagens para além da redução do tempo de processamento da operação de reamostragem. Devido ao desenvolvimento específico da solução proposta ao problema de reamostragem e redimensionamento de vídeo sem compressão segundo o modelo de cor YUV, a solução implementada não tem dependências externas de bibliotecas e ferramentas de terceiros, o que permite uma instanciação, ou *deployment*, menos complexo que a ferramenta FFmpeg. Sendo o *deployment* da solução proposta mais simples que a ferramenta FFmpeg, é possível instanciar versões da solução para a maioria dos dispositivos, desde que estes tenham o *hardware* necessário, um GPU.

Capítulo 6

Conclusões e Trabalho Futuro

A reamostragem e redimensionamento de vídeo é uma fase essencial da pós-produção de conteúdos multimédia de vídeo, pois permite a adaptação de um certo vídeo às caraterísticas de um dispositivo de reprodução ou a determinadas dimensões. Esta fase do processo de pós-produção é necessário que seja o mais rápido quanto possível de modo a que os criadores de conteúdo publiquem os mesmos aos seus utilizadores finais o quanto antes sem deteriorar a sua qualidade.

O desenvolvimento das tecnologias e arquiteturas que as unidades de processamento gráfico têm sofrido na atualidade, encaminham a utilização deste tipo de processadores a um âmbito mais genérico diferente do seu intuito inicial de processamento e renderização de conteúdo a ser visualizado numa máquina. A crescente capacidade computacional dos GPUs permite a distribuição de operações computacionais complexas entre o CPU e o GPU, e assim tirar partido de uma abordagem de computação heterogénea para a resolução de um certo problema, sendo o processo de reamostragem e redimensionamento de vídeo um desses problemas.

No trabalho desta dissertação foi desenvolvido uma ferramenta com o objetivo de diminuir o tempo de processamento despendido na aplicação do processo de reamostragem e redimensionamento de vídeo, utilizando as capacidades computacionais dos CPUs e GPUs através da ferramenta OpenMP e a plataforma de desenvolvimento CUDA. Os resultados obtidos pela solução desenvolvida neste trabalho corresponderam as expetativas iniciais do projeto, tendo sido alcançados os objetivos principais de processamento em tempo real de um vídeo em *stream* e a redução do tempo de execução da operação de reamostragem relativamente a ferramenta utilizada atualmente para o efeito e, por essa razão, a solução deste trabalho será integrada num sistema profissional de pós-produção de vídeo profissional.

A abordagem de computação heterogénea implementada do processo de reamostragem e redimensionamento de vídeo utilizando as capacidades computacionais do CPU e as unidades de processamento gráfico, resultou num ganho de performance de aproximadamente 53% face à solução que utiliza apenas o CPU.

6.1 Trabalho Realizado e Satisfação dos Objetivos

Os objetivos inicialmente estabelecidos foram cumpridos com sucesso visto que a solução proposta deste trabalho realiza o processo de reamostragem e redimensionamento de um vídeo sem compressão em tempo real para vídeos em *stream* e reduziu o tempo despendido em processamento pela ferramenta utilizada pelo proponente deste projeto, o FFmpeg.

O desenvolvimento do trabalho desta dissertação foi efetuada a partir da divisão do processo de reamostragem em duas operações: a operação de conversão do tipo de formato de pixeis e a operação de reamostragem e redimensionamento das frames de um vídeo. A divisão em duas operações permitiu a concentração dos esforços de trabalho na otimização de cada um dos problema, o que levou a uma análise mais profunda do problema.

A investigação da teoria de ambas as operações do processo de reamostragem possibilitou uma implementação mais adequada a uma arquitetura de computação paralela que se refletiu nos resultados obtidos quanto ao tempo de execução da solução. A implementação de abordagem de computação heterogénea no problema desta dissertação demonstrou que as unidades de processamento gráfico apresentam uma grande vantagem de desempenho em relação ao CPU quando se considera problemas de processamento de dados, especificamente, quando se considera o problema de processamento de vídeos através de operações baseadas em filtros.

6.2 Trabalho Futuro

No seguimento do projeto estudado durante o trabalho desta dissertação, a solução desenvolvida será integrada numa ferramenta *INGEST* profissional de pós-produção de vídeo, e a abordagem de computação heterogénea utilizando as capacidades computacionais dos CPUs e GPUs será extendida a outras fases de produção de vídeo de modo a otimizar o tempo de execução total do processo completo.

Quanto ao processo de reamostragem e redimensionamento de pós-produção estudado neste trabalho, em particular, a sua solução desenvolvida será melhorada de modo a suportar o maior número de tipos de formatos de pixeis do modelo de cor e diferentes tipos de valores de profundidade cor, como o formato V210 do modelo YUV.

A arquitetura da solução implementada será também modificada de modo a que o processamento realizado pela unidade de processamento gráfico não seja específico à operação de reamostragem e redimensionamento de frames de um vídeo, mas a suportar outras diferentes operações através de uma API.

Referências

[1] Warren Toomey. Introduction to systems architecture. Disponível em http://minnie.tuhs.org/CompArch/Lectures/week01.html, Maio 2011.

[2] Wgsimon. Transistor count and moore's law. Disponível em https://en.wikipedia.org/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2008.svg, Novembro 2008.

[3] Moisés Hernández, Ginés D Guerrero, José M Cecilia, José M García, Alberto Inuggi, Saad Jbabdi, Timothy EJ Behrens, e Stamatios N Sotiropoulos. Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using gpus. *PloS one*, 8(4):e61892, 2013.

[4] Mark Harris. Optimizing parallel reduction in cuda. nvidia dev. *Technology*, 2008.

[5] Ian Foster. *Designing and building parallel programs*, volume 78. Addison Wesley Publishing Company Boston, 1995.

[6] Blaise Barney. Openmp. *Website: https://computing. llnl. gov/tutorials/openMP*, 2008.

[7] Charles Poynton. Chroma subsampling notation. *Retrieved June*, 19:2004, 2002.

[8] Sergei Petrenko. Limitations of von neumann architecture. Em *Big Data Technologies for Monitoring of Computer Security: A Case Study of the Russian Federation*, páginas 115–173. Springer, 2018.

[9] R Michael Hord. *Parallel Supercomputing in MIMD Architectures: 0*. CRC press, 2018.

[10] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

[11] William Knight. Two heads are better than one [dual-core processors]. *IEE Review*, 51(9):32–35, 2005.

[12] Philip E Ross. Why cpu frequency stalled. *IEEE Spectrum*, 45(4), 2008.

[13] Nikola Zlatanov. Computer memory, applications and management, 02 2016.

[14] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[15] Sunpyo Hong e Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. Em *ACM SIGARCH Computer Architecture News*, volume 37, páginas 152–163. ACM, 2009.

[16] Xinxin Mei e Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[17] Mark Sutherland, Joshua San Miguel, e Natalie Enright Jerger. Texture cache approximation on gpus. Em *Workshop on Approximate Computing Across the Stack*, 2015.

[18] Cristobal A Navarro, Nancy Hitschfeld-Kahler, e Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329, 2014.

[19] David B. Skillicorn e Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, Junho 1998. URL: http://doi.acm.org/10.1145/280277.280278, doi:10.1145/280277.280278.

[20] Josep Torrellas, HS Lam, e John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[21] Fedy Abi-Chahla. Nvidia's cuda: The end of the cpu?', 2008.

[22] Kamran Karimi, Neil G Dickson, e Firas Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

[23] Shane Beers, Jim Ottaviani, Lauren White, Ann Arbor, et al. Best practices for producing quality digital video files. 2011.

[24] Charles Spence e Sarah Squire. Multisensory integration: maintaining the perception of synchrony. *Current Biology*, 13(13):R519–R521, 2003.

[25] A. Murat Tekalp. *Digital Video Processing*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2nd edição, 2015.

[26] Dave Wilson. Rgb/yuv pixel conversion. *FOURCC. org. Disponível em:< http://www. fourcc. org/fccyvrgb. php>. Acesso em*, 1, 2005.

[27] George H Joblove e Donald Greenberg. Color spaces for computer graphics. Em *ACM siggraph computer graphics*, volume 12, páginas 20–25. ACM, 1978.

[28] Ben Waggoner. *Compression for great digital video: power tips, techniques, and common sense*. Taylor & Francis, 2002.

[29] Andrew I Russell. Regular and irregular signal resampling. Relatório técnico, 2006.

[30] Zhou Wang, Alan C Bovik, Hamid R Sheikh, e Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[31] Wilhelm Burger, Mark James Burge, Mark James Burge, e Mark James Burge. *Principles of digital image processing*. Springer, 2009.

[32] Volker Rasche, Roland Proksa, R Sinkus, Peter Bornert, e Holger Eggers. Resampling of data between arbitrary grids using convolution interpolation. *IEEE transactions on medical imaging*, 18(5):385–392, 1999.

[33] Lawrence R Rabiner e Bernard Gold. Theory and application of digital signal processing. *Englewood Cliffs, NJ, Prentice-Hall, Inc., 1975. 777 p.*, 1975.

[34] Claude E Duchon. Lanczos filtering in one and two dimensions. *Journal of applied meteorology*, 18(8):1016–1022, 1979.

[35] Mei-Juan Chen, Chin-Hui Huang, e Wen-Li Lee. A fast edge-oriented algorithm for image interpolation. *Image and Vision Computing*, 23(9):791–798, 2005.

[36] Don P Mitchell e Arun N Netravali. Reconstruction filters in computer-graphics. *ACM Siggraph Computer Graphics*, 22(4):221–228, 1988.

[37] Iain E Richardson. *H. 264 and MPEG-4 video compression: video coding for next-generation multimedia*. John Wiley & Sons, 2004.

[38] Jeffrey C Mogul e Anita Borg. *The effect of context switches on cache performance*, volume 26. ACM, 1991.

[39] James Archibald e Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems (TOCS)*, 4(4):273–298, 1986.

[40] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, e Masato Edahiro. Data transfer matters for gpu computing. Em *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, páginas 275–282. IEEE, 2013.

[41] LM Itu, C Suciu, F Moldoveanu, e A Postelnicu. Comparison of single and double floating point precision performance for tesla architecture gpus. *Bull. Transilvania Univ. Brasov*, 4(53):2, 2011.

[42] V Rajaraman. Ieee standard for floating point numbers. *Resonance*, 21(1):11–30, 2016.

[43] Jean-Michel Muller. *On the definition of ulp (x)*. Tese de doutoramento, INRIA, 2005.

[44] Vern Lindberg. Uncertainties and error propagation-part i of a manual on uncertainties, graphing, and the vernier caliper. 2000.

[45] Seiji Hata, Yoshihiro Miyashita, e H Hanafusa. Human sensitivity of color defects inspection. Em *Industrial Electronics, Control, and Instrumentation, 1996., Proceedings of the 1996 IEEE IECON 22nd International Conference on*, volume 2, páginas 713–718. IEEE, 1996.

[46] Alexandre E Eichenberger, Peng Wu, e Kevin O'brien. Vectorization for simd architectures with alignment constraints. Em *Acm Sigplan Notices*, volume 39, páginas 82–93. ACM, 2004.

REFERÊNCIAS

Anexo A

Diagrama de Sequência da Solução

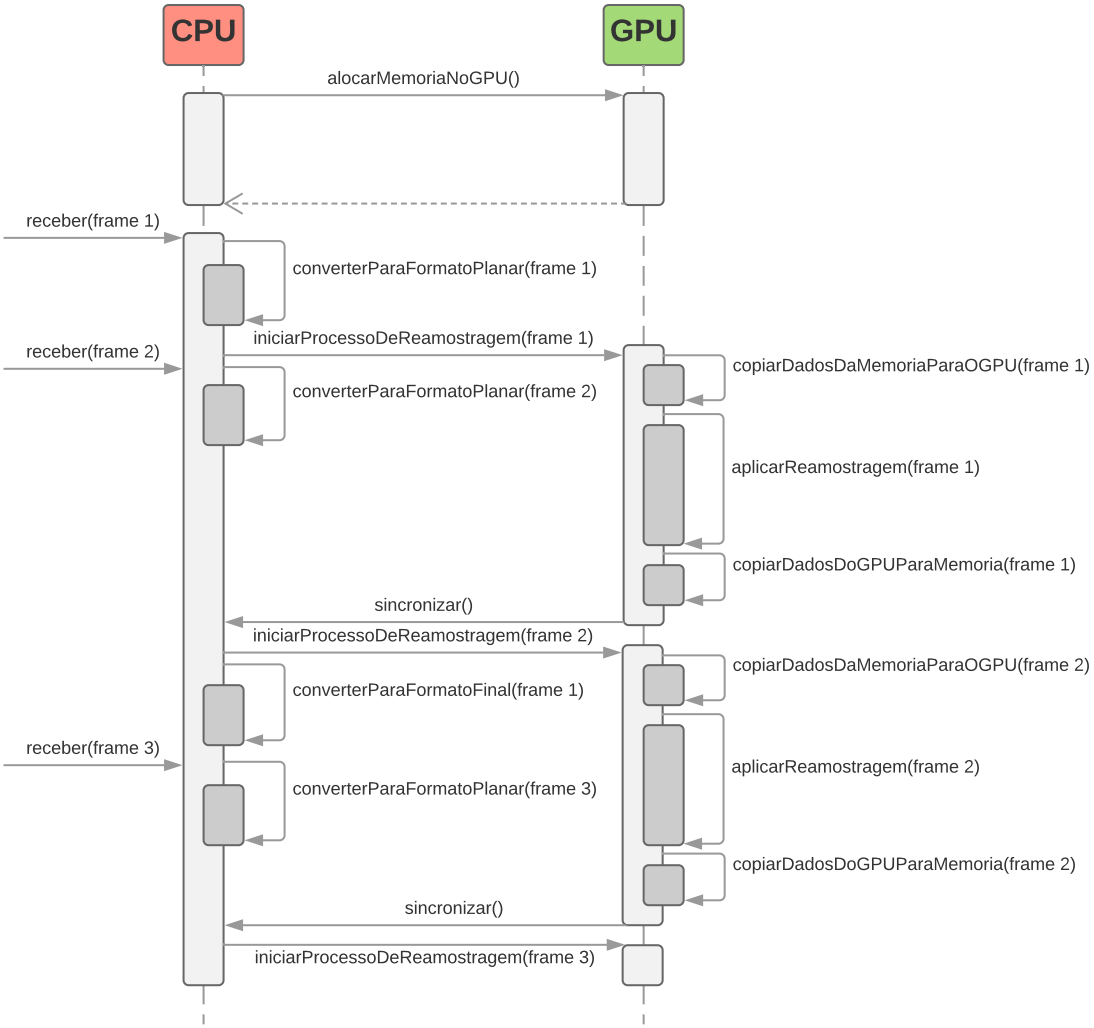

Figura A.1: Diagrama de sequência da solução.

Diagrama de Sequência da Solução

Anexo B

Tempos de Execução da Operação de Conversão de Formato de Pixeis

Tabela B.1: Tempos de execução em milissegundos da aplicação da operação de conversão de formato de pixeis em função da solução e máquina utilizada (1).

Entrada - Saída	FFmpeg M1	FFmpeg M2	Solução M1	Solução M2
UYVY - UYVY	24.097	37.659	16.084	27.149
UYVY - YUV422p	37.674	50.722	16.59	26.415
UYVY - YUV420p	32.601	43.653	13.286	20.276
UYVY - NV12	68.855	98.459	13.197	20.564
YUV422p - UYVY	30.045	38.364	15.928	27.551
YUV422p - YUV422p	24.431	34.745	16.652	26.905
YUV422p - YUV420p	41.856	57.381	14.636	19.672
YUV422p - NV12	61.519	87.206	14.692	19.367

Tabela B.2: Tempos de execução da aplicação da operação de conversão de formato de pixeis em função da solução e máquina utilizada (2).

Entrada - Saída	FFmpeg M1	FFmpeg M2	Solução M1	Solução M2
YUV420p - UYVY	29.937	37.696	15.632	27.561
YUV420p - YUV422p	47.091	63.584	16.444	26.957
YUV420p - YUV420p	18.613	25.849	12.743	20.229
YUV420p - NV12	20.812	28.392	12.438	20.108
NV12 - UYVY	86.172	119.593	16.349	26.712
NV12 - YUV422p	49.02	66.972	17.899	26.355
NV12 - YUV420p	21.265	29.275	12.578	20.179
NV12 - NV12	11.99	17.061	12.066	19.984

Anexo C

Tempos de Execução da Operação de Reamostragem e Redimensionamento

Tabela C.1: Tempos de execução em milissegundos da aplicação da operação de reamostragem e redimensionamento em função do filtro de reconstrução e unidade de processamento gráfico utilizados (2).

	Planar				Não Planar			
NN	1024x576	1280x720	2560x1440	3840x2160	1024x576	1280x720	2560x1440	3840x2160
FFmpeg	1,027	2,463	3,701	5,486	1,539	3,262	6,874	12,491
Solução GPU1	0,965	1,063	2,267	4,067	1,070	1,236	3,030	5,905
Solução GPU2	0,907	0,977	1,801	3,192	1,024	1,162	2,594	5,124
Linear								
FFmpeg	1,598	2,023	4,282	7,247	3,233	4,488	8,016	14,325
Solução GPU1	0,967	1,061	2,293	4,124	1,073	1,248	3,097	5,948
Solução GPU2	0,913	0,978	1,800	3,210	1,023	1,156	2,614	5,116
Spline								
FFmpeg	2,398	2,925	5,206	9,216	5,263	6,520	15,707	30,616
Solução GPU1	1,303	1,594	4,318	7,962	1,388	1,746	5,351	10,274
Solução GPU2	1,005	1,151	2,689	5,080	1,120	1,332	3,497	6,980